



## Informativo

### AdvPI using MVC

Model-View-Controller architecture or MVC, as it is more known, is a standard of software architecture that aims at sorting the logic of business from the logic of presentation (interface), enabling development, test and maintenance isolated from both.

Those who already developed an AdvPL application will notice that the most important different between the way to build an application in MVC and the traditional way is this sorting, which allows the use of the business rule in applications with interfaces or not, such as Web Services and automatic applications, as well as its reuse in other applications.

# Index

AdvPI using MVC.....	1
Index.....	2
1.MVC Architecture.....	6
2.Main application functions in AdvPL by using MVC.....	7
2.1 What is the ModelDef function? .....	7
2.2 What is the ViewDef function? .....	8
2.3 What is the MenuDef function? .....	8
2.4 New interface behavior .....	10
3. Applications with Browsers ( <i>FWMBrowse</i> ) .....	10
3.1 Browse Construction .....	10
3.2 Basic construction of a Browse .....	10
3.3 Browse Captions ( <i>AddLegend</i> ) .....	11
3.4 Browse Filters ( <i>SetFilterDefault</i> ) .....	12
3.5 Desabling Browse Details ( <i>DisableDetails</i> ) .....	12
3.6 Virtual fields in the Browse .....	12
3.7 Compelet example of Browse .....	13
4.Construction of AdvPL application using MVC.....	13
5.Construction of MVC application with an entity.....	14
5.1 Construction of a data structure ( <i>FWFormStruct</i> ) .....	14
5.2 Construction of the ModelDef function .....	15
5.3 Creation of a forms component in the data model ( <i>AddFields</i> ) .....	15
5.4 Description of data model components ( <i>SetDescription</i> ).....	16
5.5 ModelDef Finalization .....	16
5.6 Complete example of ModelDef .....	16
5.7 Construction of the ViewDef function .....	17
5.8 Creation of a forms component on the interface ( <i>AddFields</i> ) .....	17
5.9 Display of data on interface ( <i>CreateHorizontalBox</i> / <i>CreateVerticalBox</i> ) .....	18
5.10 Relating the interface component ( <i>SetOwnerView</i> ) .....	18
5.11 Finalization of ViewDef.....	18
5.12 Complete example of ViewDef.....	18

5.13 Finalization of creation of the application as an entity .....	19
6.Construction of MVC application with two or more entities .....	19
6.1.Construction of structures for an MVC application with two or more entities .....	20
6.2 Construction of a ModelDef function.....	20
6.3 Creation of a forms component in the data model (AddFields) .....	20
6.4 Creation of a grid component in the data model (AddFields).....	21
6.5 Creation of relations among model entities(SetRelation) .....	22
6.6 Definition of primery key (SetPrimaryKey) .....	22
6.7 Describing data model components (SetDescription) .....	22
6.8 ModelDef Finalization .....	23
6.9 Complete example of ModelDef .....	23
6.10 Construction of the ViewDef function.....	24
6.11 Creation of a forms component on the interface (AddFields) .....	24
6.12 Creation of a grid component on the interface (AddGrid).....	24
6.13 Display of data on interface (CreateHorizontalBox / CreateVerticalBox) .....	25
6.14 Setting the interface component (SetOwnerView).....	26
6.15 ViewDef Finalization.....	26
6.16 Complete example of ViewDef.....	26
6.17 Finalization of the creation of application with two or more entities .....	27
7.Treatment for data model and interface .....	28
8.Treatment for data model.....	28
8.1 Messages displayed in the interface .....	28
8.2 Obtaining data model components (GetModel).....	29
8.3 Validations .....	29
8.3.3 Validation of duplicate row (SetUniqueLine) .....	30
8.3.5 Validation of the model activation (SetVldActivate) .....	32
8.4 Manipulation of the grid component.....	32
8.4.1 Amount of grid component rows (Length).....	32
8.4.2 Go to a grid component rows (GoLine) .....	34
8.4.3 Status of the grid component row .....	34
8.4.4 Adding a grid row (AddLine).....	35
8.4.5 Deleting and recovering a grid row (DeleteLine and UnDeleteLine) .....	35

8.4.6 Permissions for a grid .....	36
8.4.7 Permission of grid with no data (SetOptional) .....	37
8.4.8 Saving and restoring the grid position (FWSaveRows / FWRestRows ) .....	37
8.4.9 Definition of maximum amount of grid rows (SetMaxLine).....	38
8.5 Obtaining and attributing values to the data model.....	38
8.6 Behavior.....	40
8.6.1 Editing component data in the data model (SetOnlyView).....	40
8.6.2 Not saving data of a component of the data model (SetOnlyQuery) .....	40
8.6.3 Obtaining the operation in execution (GetOperation).....	40
8.6.4 Data manual saving (FWFormCommit) .....	41
8.7 Completion rules (AddRules).....	42
9.Interface treatments .....	42
9.1 Incremental field (AddIncrementField) .....	43
9.2 Creation of buttons in the buttons bar (AddUserButton) .....	44
9.3 Component title (EnableTitleView) .....	45
9.4 Editing the Fields in grid components (SetViewProperty) .....	46
9.5 Creating folders (CreateFolder).....	47
9.6 Grouping fields (AddGroup) .....	49
9.7 Interface action (SetViewAction) .....	51
9.8 Interface action of the field (SetFieldAction) .....	52
9.9 Other objects (AddOtherObjects) .....	52
10.Treatment of data structure.....	56
10.1 Selection of fields for structure (FWFormStruct) .....	56
10.2 Removing structure fields (RemoveField) .....	57
10.3 Editing field properties (SetProperty) .....	57
10.4 Creation of additional fields in the structure (AddField) .....	60
10.5 Formatting the code block for structure (FWBuildFeature).....	62
10.6 Virtual MEMO type fields (FWMemoVirtual).....	63
10.7 Manual creation of trigger (AddTrigger / FwStruTrigger) .....	64
10.8 Removing the polders of a structure (SetNoFolder) .....	65
10. 9 Removing the groupings of a structure field (SetNoGroups).....	65
11.Creation of fields of total or counters (AddCalc).....	65

12. Other functions for <i>MVC</i> .....	69
12.1 Interface direct execution (FWExecView) .....	69
12.2 Active data model (FWModelActive) .....	70
12.3 Active interface (FWViewActive) .....	70
12.4 Load the data model of an existing application (FWLoadModel) .....	70
12.5 Load the interface of an existing application (FWLoadView) .....	71
12.6 Load a menu of an existing application (FWLoadMenuDef) .....	71
12.7 Obtaining standard menu (FWMVCMenu) .....	71
13. Browse with a markup column (FWMarkBrowse) .....	72
14. Multiple Browsers .....	76
15. Automatic routine .....	83
16. Entry points in <i>MVC</i> .....	96
17. Web Services for <i>MVC</i> .....	103
17.1 Web Service for data models that have an entity .....	103
17.2 Instantiation of Web Service Client .....	103
17.3 The XML structure used .....	103
17.4 Obtaining an XML structure of a data model (GetXMLData) .....	105
17.5 Entering the XML data to the Web Service .....	106
17.6 Validating the data (VldXMLData) .....	106
17.7 Validating and saving data (PutXMLData) .....	107
17.8 Obtaining the XSD scheme of a data model (GetSchema) .....	107
17.9 Complete example of Web Service .....	108
17.10 Web Services for data models with two or more entities .....	109
18. Use of New Model command .....	113
18.1 New Model Syntax .....	113
19. Reusing the existing data model or interface .....	126
19.1 We only reuse the components .....	126
19.2 Reusing and supplementing components .....	127
19.3 Complete example of an application that reuses model and interface components .....	131
Appendix A .....	133
Table of Contents .....	135

# 1.MVC Architecture

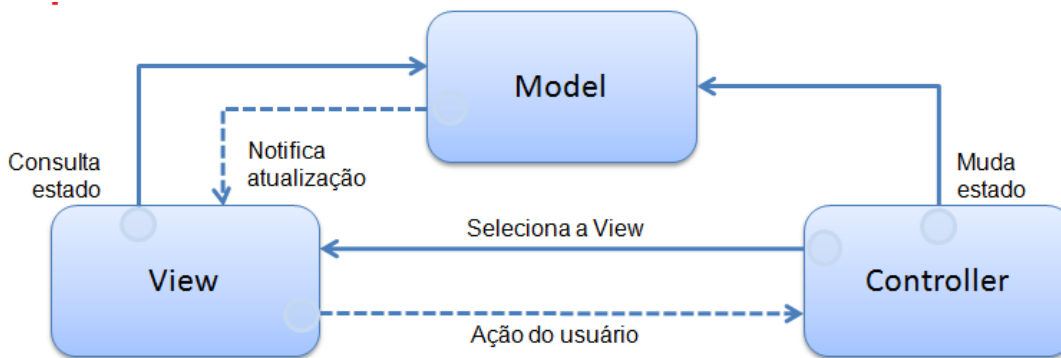
First, let us understand what MVC architecture is.

**Model-View-Controller** architecture or **MVC**, as it is more known, is a standard of software architecture that aims at sorting the logic of business from the logic of presentation (interface), enabling development, test and maintenance isolated from both.

Those who already developed an AdvPL application will notice that the most important different between the way to build an application in MVC and the traditional way is this sorting.

It allows the use of the business rule in applications with interfaces or not, such as Web Services and automatic applications, as well as its reuse in other applications.

MVC architecture has three basic components:



**Model or Data Model:** it represents the information of the application domain and provides functions to operate data, that is, it had the application's functionalities. In it, we define the business rules: tables, fields, structure, relationships etc.. The data model (Model) is also in charge of notifying the interface (View) when the information is edited.

**View or Interface:** in charge of rendering the data model (Model) and enabling users interaction, that is, in charge of displaying the data.

**Controller:** replies to users actions, enables changes in the Data Model (Model) and selects the corresponding View.

To make the development easier and faster, in the MVC implementation carried out in *AdvPL*, the developer works with the data Model (Model) definitions, the part responsible by the **Controller** is already intrinsic.

The great chance, the great paradigm to be broken when it comes to the way of thinking and developing a new application in AdvPL by using MVC is sorting the interface's business rule. For this, several new classes and methods were developed in AdvPL.

## 2. Main application functions in AdvPL by using MVC

We will now present the construction model of an application in AdvPL using MVC.

The developers in applications are in charge of defining the following functions:

**ModelDef:** It has the construction and definition of the Model (which contains business rules);

**ViewDef:** It contains the construction and definition of the View, that is, it is the construction of the interface;

**MenuDef:** It contains the definition of operations available for the data model (Model).

Each source in MVC (PRW) can only have one of each of these functions. Only one **ModelDef**, one **ViewDef** and one **MenuDef**.

After making an application in *AdvPL* using *MVC*, this application already has the following available at the end:

- **Entry Points** already available;
- A **Web Service** for your use;
- **Import or Export** XML messages.

It can be used similarly to the automatic routines of applications without MVC.

An important point in the application in MVC is that it is not based necessarily on metadata (dictionaries). As we will learn ahead, it is based on structures that can come from metadata (dictionaries) or be manually constructed.

### 2.1 What is the ModelDef function?

The ModelDef function defines the business rules where the following topics are defined:

- **All the entities** (tables) that will be part of the data model (Model);
- **Dependence rules** among the entities;
- **Validations** (of fields and application);
- **Persistence** of data (saving).

For a **ModelDef**, there is not need of an interface. As the business rule is completely separate from the interface in MVC, we can use **ModelDef** in any other application or even use a certain **ModelDef** as base for another more complex.

The **ModelDef** entities are not necessarily based on metadata (dictionaries). As we will learn further on, it is based in structures that may come from the metadata or be manually constructed.

**ModelDef** must be a **Static Function** in the application.

## 2.2 What is the ViewDef function?

The ViewDef function defines how the interface is and how the user interacts with the data model (Model) receiving information by the user, providing the data model (defined in **ModelDef**) and displaying the result.

The interface can be completely or partially based on a metadata (dictionary), allowing:

- **Reusing** the interface code as a basic interface can be added with new components;
- **Simplicity** in the development of complex interfaces. An example of this are applications in which a **GRID** depends on another. In MVC the construction of applications with dependent **GRIDS** is extremely easy;
- **Speed** in the development, creation and maintenance become much faster;
- **More than one** interface per Business Object. We can have different interfaces for each variation of a market segment, such as retail.

**ViewDef** must be a **Static Function** in the application.

## 2.3 What is the MenuDef function?

A MenuDef function defines the operations to be carried out by an application, such as **adding, editing, deleting**, etc..

It must return an array in specific format with the following information:

1. **Title;**
2. **Name of application associated;**
3. **Reserved;**
4. **Type of Transaction to be carried out.**

Which can be:

- **1 for Researching**
- **2 for Viewing**
- **3 for Adding**
- **4 for Editing**
- **5 for Deleting**
- **6 for printing**
- **7 for Copying**

5. **Access Level;**



## 6. Enabling Functional Menu;

### Example:

```
Static Function MenuDef()
Local aRotina := {}

aAdd( aRotina, { 'Visualizar', 'VIEWDEF.COMP021_MVC', 0, 2, 0, NIL } )
aAdd( aRotina, { 'Incluir' , 'VIEWDEF.COMP021_MVC', 0, 3, 0, NIL } )
aAdd( aRotina, { 'Alterar' , 'VIEWDEF.COMP021_MVC', 0, 4, 0, NIL } )
aAdd( aRotina, { 'Excluir' , 'VIEWDEF.COMP021_MVC', 0, 5, 0, NIL } )
aAdd( aRotina, { 'Imprimir' , 'VIEWDEF.COMP021_MVC', 0, 8, 0, NIL } )
aAdd( aRotina, { 'Copiar' , 'VIEWDEF.COMP021_MVC', 0, 9, 0, NIL } )

Return aRotina
```

Notice that the second parameter uses a direct call from an application. It references a *ViewDef* of a certain source (PRW).

The structure of this second parameter has the following format:

**ViewDef.<nome do fonte>**

We always mention the **ViewDef** of a source as it is the function in charge of the application's interface.

To make the development easier, in *MVC*, enter **MenuDef** as the following:

```
Static Function MenuDef()
Local aRotina := {}

ADD OPTION aRotina Title 'Visualizar' Action 'VIEWDEF.COMP021_MVC' OPERATION 2 ACCESS 0
ADD OPTION aRotina Title 'Incluir' Action 'VIEWDEF.COMP021_MVC' OPERATION 3 ACCESS 0
ADD OPTION aRotina Title 'Alterar' Action 'VIEWDEF.COMP021_MVC' OPERATION 4 ACCESS 0
ADD OPTION aRotina Title 'Excluir' Action 'VIEWDEF.COMP021_MVC' OPERATION 5 ACCESS 0
ADD OPTION aRotina Title 'Imprimir' Action 'VIEWDEF.COMP021_MVC' OPERATION 8 ACCESS 0
ADD OPTION aRotina Title 'Copiar' Action 'VIEWDEF.COMP021_MVC' OPERATION 9 ACCESS 0

Return aRotina
```

The final result is the same, the difference is only the form of construction. **We recommend the second form** that uses the command format and not a vector's position, as an occasional maintenance is easier.

**MenuDef** must be a **Static Function** within the application.

Using the **FWMVCMenu** function, you obtain a standard menu with the following options: Viewing, Adding, Editing, Deleting, Printing and Copying. It must be transferred as parameter in the source name.

For example:

```
Static Function MenuDef()
```

```
Return FWMVCMenu( "COMP021_MVC" )
```

This created a **Menudef** exactly as the previous example. For more details, see chapter **12.7 Obtain a standard menu (FWMVCMenu)**.

## 2.4 New interface behavior

In the applications developed in traditional *AdvPL*, after the conclusion of an editing operation, the interface closes and returns to the Browse.

In the applications in MVC, after the adding and editing operations, the interface remains active and a message of successful operation is displayed on the bottom.

## 3. Applications with Browsers (FWMBrowse)

To construct an application that has a **Browse**, MVC uses **FWMBrowse** class.

This class displays a **Browse** object that is constructed from the metadata (dictionaries).

This class **was not developed exclusively** for MVC, applications that are in MVC can also use it. In MVC, we will be using it.

Its **characteristics** are:

- Replacing Browse components;
- Reducing maintenance time in the event of addition of a new requirement;
- Being independent from the Microsiga Protheus Environment

The **main improvements** are:

- Standardization of color caption;
- Better usability when treating filters;
- Standards of color, fonts and caption defined by the user – Visually impaired;
- Reduction of the number of operations in SGBD (at least 3 times fast);
- New visual default.

### 3.1 Browse Construction

We will be discussing the main functions and characteristics to for the use in applications with MVC.

### 3.2 Basic construction of a Browse

We will begin the basic construction of a Browse

First create a Browse object as follows:

```
oBrowse := FWMBrowse():New()
```

We defined the table to be displayed on the Browse by using the **SetAlias** method. The columns, orders, etc..

The display is obtained by the metadata (dictionaries).

```
oBrowse:SetAlias('ZA0')
```

We defined the title to be displayed as **SetDescription** method.

```
oBrowse:SetDescription('Cadastro de Autor/Interprete')
```

And in the end the activate the class.

```
oBrowse:Activate()
```

With this basic structure, we built an application with Browse.

The Browse presented automatically already has:

- Registration search;
- Configurable filter;
- Setting of columns and display;
- Printing.

### 3.3 Browse Captions (**AddLegend**)

To use the Browse captions, we use the **AddLegend** method, which has the following synthax:

```
AddLegend( <cRegra>, <cCor>, <cDescrição> )
```

#### **Example:**

```
oBrowse:AddLegend( "ZA0_TIPO=='1'", "YELLOW", "Autor" )
```

```
oBrowse:AddLegend( "ZA0_TIPO=='2'", "BLUE" , "Interprete" )
```

**cRegra** is the expression in *AdvPL* to define the caption.

**cCor** is the parameter to define the color of each caption item.

#### **The following values are possible:**

GREEN	For the color Green
RED	For the color Red
YELLOW	For the color Yellow
ORANGE	For the color Orange
BLUE	For the color Blue
GRAY	For the color Gray
BROWN	For the color Brown
BLACK	For the color Black
PINK	For the color Pink
WHITE	For the color White

**cDescrição** is displayed for each caption item.

**Note:**

- Each caption becomes a filter option automatically.
- Careful while defining the caption's rules. If there are rules in conflict, the caption corresponding to the first rules satisfied is displayed.

### ***3.4 Browse Filters (SetFilterDefault)***

To define a Browse filter, we use the **SetFilterDefault** method, which has the following syntax:

```
SetFilterDefault ( <filtro> )
```

**Example:**

```
oBrowse:SetFilterDefault( "ZA0_TIPO=='1'" )
```

or

```
oBrowse:SetFilterDefault( "Empty(ZA0_DTAFAL)" )
```

The filter expression is in AdvPL.

The filter defined in the application does not annul the possibility of the user to make his/her own filters. The filters made by the user are applied with what is defined in the application (**AND** conditions).

**Example:**

If it is defined in the application that only customers who are legal entities are displayed, the user can create a filter to display only customers in the State of São Paulo so that legal entities of the State of São Paulo are displayed. The user filter was executed and the application's original filter was respected.

**Note:** The application's filter cannot be disabled by the user.

### ***3.5 Desabling Browse Details (DisableDetails)***

The data of the positioned row are displayed automatically for the **Browse**. To disable this feature, we use the **DisableDetails** method.

**Example:**

```
oBrowse:DisableDetails()
```

### ***3.6 Virtual fields in the Browse***

Usually, to display the virtual fields in Browsers, we can use the **Position** function.

In the new Browse, this practice is even more important. When it finds the **Position** function in the definition of a virtual field and the database is **SGBD** (using **TOTVSDbAccess**), the Browse adds an **INNER JOIN** in the query to be sent to **SGBD**, therefore improving the performance for the data extraction.

Therefore, always use the **Position** function to display virtual fields.

### 3.7 Compelet example of Browse

```
User Function COMP011_MVC()
Local oBrowse
// Instanciamento da Classe de Browse
oBrowse := FWMBrowse():New()

// Definição da tabela do Browse
oBrowse:SetAlias('ZA0')

// Definição da legenda
oBrowse:AddLegend( "ZA0_TIPO=='1'", "YELLOW", "Autor" )
oBrowse:AddLegend( "ZA0_TIPO=='2'", "BLUE" , "Interprete" )

// Definição de filtro
oBrowse:SetFilterDefault( "ZA0_TIPO=='1'" )

// Título da Browse
oBrowse:SetDescription('Cadastro de Autor/Interprete')

// Opcionalmente pode ser desligado a exibição dos detalhes
//oBrowse:DisableDetails()

// Ativação da Classe
oBrowse:Activate()

Return NIL
```

## 4.Construction of AdvPL application using MVC

Now we start the construction of the MVC part of the application, that is, the **ModelDef** functions which include business rules, and **ViewDef** that includes the interface.

An **important** point that must be noticed is that, just as **MenuDef**, there can **only be one ModelDef function** and one **ViewDef function** in a source code.

If you must work in more than one data model (Model) in a certain situation, the application must be broken into several source codes (PRW), each with only on **ModelDef** and one **ViewDef**.

## 5. Construction of MVC application with an entity

We will show how to create a MVC application with only one entity involved.

### 5.1 Construction of a data structure (*FWFormStruct*)

The first thing we must do is to create the structure used in the data model (Model).

Structures are objects that contain definitions of data necessary to use **ModelDef** or **ViewDef**.

These purposes contain:

- Structure of the Fields;
- Indexes
- Triggers;
- Filling out rules (we will see this ahead);
- Etc.

As we previously said, MVC does not work linked to Microsiga Protheus metadata (dictionaries). It works linked to structures. Then these structures can be built from metadata.

With the **FWFormStruct** function, the structure is created from the metadata.

Its syntax is:

```
FWFormStruct( <nTipo>, <cAlias> )
```

Where:

**nTipo**            Type of structure construction: 1 for data model (Model) and 2 for interface (View);

**cAlias**            Alias of the table in metadata;

#### **Example:**

```
Local oStruZA0 := FWFormStruct( 1, 'ZA0' )
```

In the example, the **oStruZA0** object is a structure for use in a data model (Model). The first parameter (1) indicates that the structure is for use in the model and the second parameter indicates what metadata table is used for structure creation (ZA0).

```
Local oStruZA0 := FWFormStruct( 2, 'ZA0' )
```

In the example given, the **oStruZA0** object is a structure for use in an interface (View). The first parameter (2) indicates that the structure is for use in an interface and the second parameter indicates what metadata table is used for structure creation (ZA0).

Farther ahead, we will learn how to create structures manually and how to select the fields that are part of the structures and other specific treatments of the structure.

**Important:** For the data model (*Model*), the **FWFormStruct** function brings all fields composing the table to the entire structure, regardless of the level, use or module. It also considers virtual fields.

For the interface (*View*), the **FWFormStruct** function brings the fields according to the level, use or module to the structure.

## 5.2 Construction of the ModelDef function

As it has been previously stated, in this function, the business rules are defined, or data model (*Model*).

It has the definitions of:

- Entities involved;
- Validations;
- Relationships;
- Data persistence (saving);
- Etc.

We start the **ModelDef** function:

```
Static Function ModelDef()
Local oStruZA0 := FWFormStruct( 1, 'ZA0' )
Local oModel // Modelo de dados que será construído
```

Building the Model

```
oModel := MPFormModel():New( 'COMP011M' )
```

**MPFormModel** is the class used for the construction of a data model (*Model*) object.

We must give an identifier (*ID*) for the model as a whole and also for each component.

This is an MVC featur. Every model or interface component must have an ID, such as forms, **GRIDs**, **boxes**, etc.

**COMP011M** is the identifier (*ID*) given to Model. It is important to emphasize comparisons with the Model identifier (*ID*):

- If an application is a **Function**, the data model (*Model*) identifier (*ID*) can have the same name of the main function and this practice is recommended to facilitate the codification. For example, if we are writing the XPTO function, the data model (*Model*) identifier (*ID*) can be XPTO.
- If the application is a **User Function**, the data model (*Model*) identifier (*ID*) **CANNOT** have the same name of the main function because of the entry points that are created automatically when we develop an MVC application. This will be detailed further on (see chapter **16.Entry points in MVC** ).

## 5.3 Creation of a forms component in the data model (AddFields)

The **AddFields** adds a form component to the model.

The data model (Model) structure must start with a form component.

**Example:**

```
oModel:AddFields( 'ZAOMASTER', /*cOwner*/, oStruZA0 )
```

We must give an identifier (ID) for each model component.

**ZAOMASTER** is the identifier (ID) given to the form component in the model, **oStruZA0** is the structure used in the form and was previously constructed using **FWFormStruct**. Notice that the second parameter (owner) was not informed, this is because this first model component is the data model's (Model) **Parent** so it does not have a superior component or **owner**.

## 5.4 Description of data model components (SetDescription)

Always defining a description for the models components.

With the **SetDescription** method, we add a description to the data model (Model). This description is used in several places such as Web Services, for example.

We added the description of the **data model**:

```
oModel:SetDescription( 'Modelo de dados de Autor/Interprete' )
```

We added the description of the **data model components**:

```
oModel:GetModel( 'ZAOMASTER' ):SetDescription( 'Dados de Autor/Interprete' )
```

For a model containing only one component, it seems redundant to give a description for a data model (Model) as a whole and one for the component, but this action becomes clearer when we study other models with more than one component.

## 5.5 ModelDef Finalization

At the end of the **ModelDef** function, the data model (Model) object generated in the function must be returned.

```
Return oModel
```

## 5.6 Complete example of ModelDef

```
Static Function ModelDef()  
  
// Cria a estrutura a ser usada no Modelo de Dados  
Local oStruZA0 := FWFormStruct( 1, 'ZA0' )  
Local oModel // Modelo de dados que será construído  
  
// Cria o objeto do Modelo de Dados  
oModel := MPFormModel():New('COMP011M' )
```



```
// Adiciona ao modelo um componente de formulário
oModel:AddFields( 'ZAOMASTER', /*cOwner*/, oStruZA0)

// Adiciona a descrição do Modelo de Dados
oModel:SetDescription( 'Modelo de dados de Autor/Interprete' )

// Adiciona a descrição do Componente do Modelo de Dados
oModel:GetModel( 'ZAOMASTER' ):SetDescription( 'Dados de Autor/Interprete' )

// Retorna o Modelo de dados
Return oModel
```

## 5.7 Construction of the ViewDef function

The interface (View) is in charge of rendering the data model (Model) and enabling users interaction, that is, in charge of displaying the data.

**ViewDef** includes the definition of every visual part of the application.

**We start the function:**

```
Static Function ViewDef()
```

The interface (View) always work based on a data model (*Model*). We will create one data model object in the desired **ModelDef**.

With the **FWLoadModel** function we obtain the data model (Model) defined in a source, but nothing would prevent the use of the model from any other source in MVC. With this, we can reuse the same data model (Model) in more than one interface (View).

```
Local oModel := FWLoadModel( 'COMP011_MVC' )
```

**COMP011\_MVC** is the name of the source in which we want to obtain the data model (Model).

**Starting construction of interface (View)**

```
oView := FWFormView():New()
```

**FWFormView** is the class to be used for the construction of an interface (View) object.

We defined what data model (Model) will be used on the interface (Model).

```
oView:SetModel( oModel )
```

## 5.8 Creation of a forms component on the interface (AddFields)

We added a form type control (old **Enchoice**) to our interface (View), so we use the **AddField** method

The interface (View) must start with a type-form component.

```
oView:AddField( 'VIEW_ZA0', oStruZA0, 'ZAOMASTER' )
```

We must give an identifier (ID) for each interface (View) component.

**VIEW\_ZA0** is the identifier (ID) given to the interface (View) component, **oStruZA0** is the structure to be used and **ZAOMASTER** is the identifier (ID) of the data model (Model) components linked to this interface (View) component.

Each interface (View) component must have a related data model (Model) component. This is the same as saying that the **ZAOMASTER** data will be displayed on the interface (View) in **VIEW\_ZA0** component

## **5.9 Display of data on interface (CreateHorizontalBox / CreateVerticalBox)**

We must always create a **container**<sup>1</sup>, an object, to receive some interface (View) element. In MVC, we will always create a horizontal or vertical **box** for this.

A method to create the horizontal **box** is:

```
oView>CreateHorizontalBox( 'TELA' , 100 )
```

We must give an identifier (ID) for each interface (View) component.

**SCREEN** is the identifier (ID) given to the **box** and the number **100** represents the percentage of the screen to be used by the Box.

There are no references to absolute screen guidelines in MVC. All visual components are always **All Client**, that is, they will take up all the **container** where they are inserted

## **5.10 Relating the interface component (SetOwnerView)**

We have to relate the interface (View) component with a **box** to display, so we use the **SetOwnerView** method.

```
oView:SetOwnerView( 'VIEW_ZA0', 'TELA' )
```

This way, the **VIEW\_ZA0** component will be displayed on the screen by using the box **SCREEN**.

## **5.11 Finalization of ViewDef**

At the end of the **ViewDef** function, the interface (View) object generated must be returned.

```
Return oView
```

## **5.12 Complete example of ViewDef**

```
Static Function ViewDef()
```

```
// Cria um objeto de Modelo de dados baseado no ModelDef() do fonte informado
```

---

<sup>1</sup> A certain area defined by the developer to group visual components, for example, Panel, Dialog, Window, etc

```

Local oModel := FWLoadModel( 'COMP011_MVC' )

// Cria a estrutura a ser usada na View
Local oStruZA0 := FWFormStruct( 2, 'ZA0' )

// Interface de visualização construída
Local oView
// Cria o objeto de View
oView := FWFormView():New()

// Define qual o Modelo de dados será utilizado na View
oView:SetModel( oModel )
// Adiciona no nosso View um controle do tipo formulário
// (antiga Enchoice)
oView:AddField( 'VIEW_ZA0', oStruZA0, 'ZAOMASTER' )

// Criar um "box" horizontal para receber algum elemento da view
oView:CreateHorizontalBox( 'TELA' , 100 )

// Relaciona o identificador (ID) da View com o "box" para exibição
oView:SetOwnerView( 'VIEW_ZA0', 'TELA' )

// Retorna o objeto de View criado
Return oView

```

### 5.13 Finalization of creation of the application as an entity

We create a AdvPL application using MVC in which there is only one entity involved.

- We constructed **ModelDef**;
- We constructed **ViewDef**.

This application would be the equivalent to the application of **Model1** type that is normally made.

Next, we will learn about the construction of applications using two or more entities.

## 6. Construction of MVC application with two or more entities

So far, we learned about the construction of an application that used only one entity. We will learn about the construction two or more entities.

The construction of the application follows the same steps we saw so far: Construction of **ModelDef** and **ViewDef**. The basic difference is that now each one of them has more than one

component and they relate.

## **6.1. Construction of structures for an MVC application with two or more entities**

As we described, the first thing we must do is to create the structure used in the data model (Model). We have to create a structure for each entity that is a part of the model. In the event of 2 entities, 2 structures, in the event of 3 entities, 3 structures and so on.

We will show an application with 2 entities in a **Master-Detail** dependence relationship, for example, a Sale Order with the order header would be the **Master** and the items would be the **Detail**

The construction of the structures would be:

```
Local oStruZA1 := FWFormStruct( 1, 'ZA1' )
Local oStruZA2 := FWFormStruct( 1, 'ZA2' )
```

In the previous example, the **oStruZA1** object is a structure to be used in a data model (Model) for the **Master** entity and **oStruZA2** for the **Detail** entity.

The first parameter (1) indicates that the structure is to be used in a data model (Model) and the second indicates what table is used for structure creation.

```
Local oStruZA1 := FWFormStruct( 2, 'ZA1' )
Local oStruZA2 := FWFormStruct( 2, 'ZA2' )
```

In the previous example, the **oStruZA1** object is a structure to be used in an interface (View) for the **Master** entity and **oStruZA2** for the **Detail** entity. The first parameter (2) indicates that the structure is to be used in an interface (View) and the second indicates what table is used for structure creation.

## **6.2 Construction of a ModelDef function**

We start the **ModelDef** function:

```
Static Function ModelDef()
Local oStruZA1 := FWFormStruct( 1, 'ZA1' )
Local oStruZA2 := FWFormStruct( 1, 'ZA2' )
Local oModel // Modelo de dados que será construído
```

Notice that 2 structures were created in the code, one for each entity.

We started the Model construction

```
oModel := MPFormModel():New( 'COMP021M' )
```

We must give an identifier (*ID*) for the data model (Model) and for each Model component.

**COMP021M** is the identifier (*ID*) given to the data model (Model).

## **6.3 Creation of a forms component in the data model (AddFields)**

The **AddFields** method adds a form component to the model.

The model structure must start with a form component.

```
oModel:AddFields( 'ZA1MASTER', /*cOwner*/, oStruZA1 )
```

We must give an identifier (ID) for each model component.

**ZA1MASTER** is the identifier (ID) given to the form in the model, **oStruZA1** is the structure used in the form and was previously constructed using **FWFormStruct**. Notice that the second parameter (owner) was not informed, this is because this first model component is the data model's (Model) **Parent** so it does not have a superior component or **owner**.

## 6.4 Creation of a grid component in the data model (AddFields)

The relationship of dependence between the entities is **Master-Detail**, that is, there is 1 **Parent** occurrence for **n Secondary** occurrences (1-*n*)

When an entity occurs **n** times in the model in comparison to the other, we must define a **Grid** component for this entity.

The **AddGrid** method adds a grid component to the model.

```
oModel:AddGrid( 'ZA2DETAIL', 'ZA1MASTER', oStruZA2 )
```

We must give an identifier (ID) for each model component.

**ZA2DETAIL** is the identifier (ID) given to the component in the model, **oStruZA2** is the structure used in the form and was previously constructed using **FWFormStruct**. Notice that the second parameter (owner) was informed this time because this entity depends on the first (Master), so **ZA1MASTER** is the component superior to or **owner** of **ZA2DETAIL**.

## 6.5 Creation of relations among model entities (SetRelation)

In the model, we must list all the entities that are a part of it. In our example, we have to list the **Detail** entity with the **Master** entity.

A very simple rule to understand this is: Every model entity that has a superior (owner) must have its relationship defined to it. In other words, you must say what keys are for the secondary relation and what keys are for the parent relation.

The method used for this definition is **SetRelation**.

### Example:

```
oModel:SetRelation( 'ZA2DETAIL', { { 'ZA2_FILIAL', 'xFilial( "ZA2" )' }, { 'ZA2_MUSICA', 'ZA1_MUSICA' } }, ZA2->( IndexKey( 1 ) ) )
```

**ZA2DETAIL** is the **Detail** entity identifier (*ID*), the second parameter is a bi-dimensional vector where relations between each field of secondary entity and Parent entity are defined. The third parameter is the order of this data in the component.

What we are saying in the example above is that the relation of the Detail entity is **ZA2\_FILIAL** and **ZA2\_MUSICA**, the **ZA2\_FILIAL** value is given by **xFilial()** and **ZA2\_MUSICA** comes from **ZA1\_MUSICA**.

**Note:** The relationship is always defined from **Detail (Secondary)** to **Master (Parent)**, in the identifier (*ID*) and in the folder of the bi-dimensional vector.

## 6.6 Definition of primery key (SetPrimaryKey)

The data model must know what the primary key is for the main entity of data model (Model).

If the structure was constructed, use **FWFormStruct**, the primary key is defined in the metadata (dictionaries).

If the structured was constructed manually or if the entity does not have a definition of a single key in the metadata, we must define this key with the **SetPrimaryKey** method.

### Example:

```
oModel: SetPrimaryKey( { "ZA1_FILIAL", "ZA1_MUSICA" } )
```

Where the passed parameter is a vector with the fields composing the primary key. **Use this method only if needed.**

Always define the primary key for the model. If you are not able to create a primary key for the main entity, enter it in the data model as the following:

```
oModel: SetPrimaryKey( {} )
```

## 6.7 Describing data model components (SetDescription)

Defina sempre uma descrição para os componentes do modelo. With the **SetDescription** method, we add a description to the data model. This description is used in several places such as Web Services, for example.

We added the data model description.

```
oModel:SetDescription( 'Modelo de Musicas' )
```

We added the description of the data model components.

```
oModel:GetModel( 'ZA1MASTER' ):SetDescription( 'Dados da Musica' )
oModel:GetModel( 'ZA2DETAIL' ):SetDescription( 'Dados do Autor Da Musica' )
```

Notice that this time we defined a description for the model and one for each model component.

## 6.8 ModelDef Finalization

At the end of the **ModelDef** function, the data model (Model) object generated in the function must be returned.

```
Return oModel
```

## 6.9 Complete example of ModelDef

```
Static Function ModelDef()

// Cria as estruturas a serem usadas no Modelo de Dados
Local oStruZA1 := FWFormStruct( 1, 'ZA1' )
Local oStruZA2 := FWFormStruct( 1, 'ZA2' )
Local oModel // Modelo de dados construído

// Cria o objeto do Modelo de Dados
oModel := MPFormModel():New( 'COMP021M' )

// Adiciona ao modelo um componente de formulário
oModel:AddFields( 'ZA1MASTER', /*cOwner*/, oStruZA1 )

// Adiciona ao modelo uma componente de grid
oModel:AddGrid( 'ZA2DETAIL', 'ZA1MASTER', oStruZA2 )

// Faz relacionamento entre os componentes do model
oModel:SetRelation( 'ZA2DETAIL', { { 'ZA2_FILIAL', 'xFilial( "ZA2" )' }, { 'ZA2_MUSICA',
'ZA1_MUSICA' } }, ZA2->( IndexKey( 1 ) ) )

// Adiciona a descrição do Modelo de Dados
oModel:SetDescription( 'Modelo de Musicas' )

// Adiciona a descrição dos Componentes do Modelo de Dados
oModel:GetModel( 'ZA1MASTER' ):SetDescription( 'Dados da Musica' )
oModel:GetModel( 'ZA2DETAIL' ):SetDescription( 'Dados do Autor Da Musica' )
```

```
// Retorna o Modelo de dados
Return oModel
```

## 6.10 Construction of the ViewDef function

We started a function.

```
Static Function ViewDef()
```

The interface (View) always works based on a data model (*Model*).

We created one data model object in the desired **ModelDef**.

With the **FWLoadModel** function we obtain the data model (Model) defined in a source, in our case it is the source itself, but nothing would prevent the use of the data mode (Model) from any other source in MVC. With this, we can reuse the same data model (Model) in more than one interface (View).

```
Local oModel := FWLoadModel( 'COMP021_MVC' )
```

**COMP021\_MVC** is the name of the source in which we want to obtain the model.

We started the construction of interface (View)

```
oView := FWFormView():New()
```

**FWFormView** is the class to be used for the construction of an interface (View) object.

We defined what data model (Model) will be used on the interface (View).

```
oView:SetModel( oModel )
```

## 6.11 Creation of a forms component on the interface (AddFields)

We added a form type control (old **Enchoice**) to our interface (View), so we use the **AddField** method

The interface (View) must start with a form component.

```
oView:AddField( 'VIEW_ZA1', oStruZA1, 'ZA1MASTER' )
```

We must give an identifier (ID) for each interface (View) component. **VIEW\_ZA1** is the identifier (ID) given to the interface (View) component, **oStruZA1** is the structure to be used and **ZA1MASTER** is the identifier (ID) of the data model (Model) components linked to this interface (View) component.

Each interface (View) component must have a related data model (Model) component. This is the same as saying that the **ZA1MASTER** data will be displayed on the interface (View) in **VIEW\_ZA1** component.

## 6.12 Creation of a grid component on the interface (AddGrid)

We added a grid type control (old **GetDados**) to our interface (View), so we use the **AddGrid** method.

```
oView:AddGrid( 'VIEW_ZA2', oStruZA2, 'ZA2DETAIL' )
```

We must give an identifier (ID) for each interface (View) component.



**VIEW\_ZA2** is the identifier (ID) given to the interface (View) component, **oStruZA2** is the structure to be used and **ZA2MASTER** is the identifier (ID) of the data model (Model) components linked to this interface (View) component.

Each interface (View) component must have a related data model (Model) component. This is the same as saying that the **ZA2DETAIL** data will be displayed on the interface (View) in **VIEW\_ZA2** component.

**Note:** Note that we do not mention that the entity is superior because this function is the data model's. The interface (View) only reflects the model data.

### **6.13 Display of data on interface (CreateHorizontalBox / CreateVerticalBox)**

We must always create a container, an object, to receive some interface (View) element.

In *MVC*, we will always create a horizontal or vertical **box** for this.

The method to create the horizontal box is:

```
oView.CreateHorizontalBox( 'SUPERIOR', 15 )
```

We must give an identifier (ID) for each interface (View) component. **SUPERIOR** is the identifier (ID) given to the **box** and the number **15** represents the percentage of the screen to be used by the **box**.

As we have two components, we must define more than one box for the second component

```
oView.CreateHorizontalBox( 'INFERIOR', 85 )
```

**INFERIOR** is the identifier (ID) given to the **box** and the number **85** represents the percentage of the screen to be used by the **box**.

**Note:** The sum of percentage of the boxes of the same level must always be 100%.

## 6.14 Setting the interface component (SetOwnerView)

We have to relate the interface (View) component with a **box** to display, so we use the **SetOwnerView** method.

```
oView:SetOwnerView( 'VIEW_ZA1', 'SUPERIOR' )  
oView:SetOwnerView( 'VIEW_ZA2', 'INFERIOR' )
```

This way, the **VIEW\_ZA1** component is displayed on the screen by the **SUPERIOR** box and the **VIEW\_ZA2** component is displayed on the screen by the **INFERIOR** box.

**Note:** Notice that the **Parent** entity's data take 15% of the screen and the **Secondary** entity 85%, as:

Model ID	View ID	Box ID
ZA1MASTER	VIEW_ZA1	SUPERIOR (15%)
ZA2DETAIL	VIEW_ZA2	INFERIOR (85%)

## 6.15 ViewDef Finalization

At the end of the **ViewDef** function, the interface (View) object generated must be returned.

Return oView

## 6.16 Complete example of ViewDef

```
Static Function ViewDef()  
  
// Cria um objeto de Modelo de dados baseado no ModelDef do fonte informado  
Local oModel := FWLoadModel( 'COMP021_MVC' )  
  
// Cria as estruturas a serem usadas na View  
Local oStruZA1 := FWFormStruct( 2, 'ZA1' )  
Local oStruZA2 := FWFormStruct( 2, 'ZA2' )  
  
// Interface de visualização construída  
Local oView  
  
// Cria o objeto de View  
oView := FWFormView():New()  
  
// Define qual Modelo de dados será utilizado  
oView:SetModel( oModel )
```

```
// Adiciona no nosso View um controle do tipo formulário (antiga Enchoice)
oView:AddField( 'VIEW_ZA1', oStruZA1, 'ZA1MASTER' )

//Adiciona no nosso View um controle do tipo Grid (antiga Getdados)
oView:AddGrid( 'VIEW_ZA2', oStruZA2, 'ZA2DETAIL' )

// Cria um "box" horizontal para receber cada elemento da view
oView:CreateHorizontalBox( 'SUPERIOR', 15 )
oView:CreateHorizontalBox( 'INFERIOR', 85 )

// Relaciona o identificador (ID) da View com o "box" para exibição
oView:SetOwnerView( 'VIEW_ZA1', 'SUPERIOR' )
oView:SetOwnerView( 'VIEW_ZA2', 'INFERIOR' )

// Retorna o objeto de View criado
Return oView
```

## 6.17 Finalization of the creation of application with two or more entities

We create an AdvPL application using MVC in which there are two entities involved.

- We constructed **ModelDef**;
- We constructed **ViewDef**.

This application would be the equivalent to **Model3** type applications that are usually made.

If the need is to construct one application with more than 2 entities, the process is the same as for 2. The difference is only the amount of each component or object created.

For the data model (Model), if the application has 3 entities, you need 3 structures, 3 **AddFields** or **AddGrid** components and 2 relations. If the application has 4 entities, you need 4 structures, 4 **AddFields** or **AddGrid** components and 3 relations, and so on.

For the interface (View), if the application has 3 entities, you need 3 structures, 3 **AddField** or **AddGrid** components and 3 boxes. If the application has 4 entities, you need 4 structures, 4 **AddField** or **AddGrid** components and 4 boxes, and so on.

The data model and the interface grow to the extent of the amount of entities related. However, the basic form of construction is always the same.

## 7.Treatment for data model and interface

Now we already know how to construct an application in *MVC* using *n* entities. In this chapter, we will display the specific treatments for some needs in the construction of an application for business rules and the interface, as the idea is always the same in terms of hierarchy.

### **Example:**

- Validations;
- Permissions;
- Transfer in rows;
- Obtain and attribute values;
- Persistence of data;
- Create buttons;
- Create folders; etc.

## 8.Treatment for data model

We will see some treatments that can be carried out in the data model (Model) according to need:

- Validations;
- Behaviors;
- Manipulation.
- Obtain and attribute values to the data model (Model);
- Saving data manually;
- Completion rules.

### **8.1 Messages displayed in the interface**

Messages are used especially during the validations made in the data model.

**Let us analyze:** A basic MVC point is the sorting of the business rule from the interface.

The validation is a process executed in a business rule and an occasional error message displayed to the user. It is a process that must be executed in the interface, that is, it cannot be executed in the business rule.

To work on this situation, a treatment for the **Help** function was made.

The **Help** function can be used in the functions in the data model (Model), but MVC saves these messages and it is only displayed when the control returns to the interface.

For example, a certain function includes:

```
If nPrcUnit == 0 // Preço unitário
    Help( ,, 'Help',,, 'Preço unitário não informado.', 1, 0 )
EndIf
```

Suppose the error message was enabled because the unit price is zero (0), nothing is displayed to the user in that moment. You can notice that by debugging the source. You will notice that by passing the **Help** function, nothing happens but when the internal control returns to the interface, the message is displayed.

This training was made only for the **Help** function. Functions such as **MsgStop**, **MsgInfo**, **MsgYesNo**, **Alert**, **MostraErro**, etc. **cannot** be used.

## 8.2 Obtaining data model components (GetModel)

During the development, many times we have to handle the data model (Model). To make this handling easier, we can work with a specific part (one component) at a time, instead of working with the entire model.

To do so, you use the **GetModel** method.

```
Local oModelZA2 := oModel:GetModel( 'ZA2DETAIL' )
```

**oModelZA2** is the object that contains a data model (Model) component and **ZA2DETAIL** is the identifier (ID) of the component we want.

If we have part of the data model (Model) and want to get the complete model, we can also use **GetModel**.

```
Local oModel := oModelZA2:GetModel()
```

**oModel** is the containing the complete data model (Model).

## 8.3 Validations

Within the existing data model, several points where the validations necessary to business rules can be inserted. The data model (Model) as a whole has its points and each model component too.

### 8.3.1 Post-validation of the model

It is the validation carried out after completing the data model (Model) and its confirmation. It would be the equivalent to the old **EverythingOk** process.

The data model (Model) already performs the validation if the mandatory fields of all model components are filled out. This validation is executed after this.

We defined the data model (Model) post-validation as a code block in the 3rd parameter of construction of the **MPFormModel** model.

```
oModel := MPFormModel():New( 'COMP011M', , { |oModel| COMP011POS( oModel ) } )
```

The code block receives an object as parameter. It is the model which can be passed to the function performing the validation.

```
Static Function COMP011POS( oModel )

Local lRet := .T.
Local nOperation := oModel:GetOperation
    // Segue a função ...
Return lRet
```

The function called by the code block must return a logical value where the operation is performed if .T. (true) and not performed if .F. (false).

### 8.3.2 Post-validation of row

In a data model (Model) in which grid components are, a validation to be executed in the exchange of grid lines can be defined. It would be the equivalent to the old **RowOk** process.

We define the post-validation of row as a code block in the 5th parameter of the **AddGrid** method.

```
oModel:AddGrid( 'ZA2DETAIL', 'ZA1MASTER', oStruZA2, , { |oModelGrid| COMP021LPOS(oModelGrid) } )
```

The code block receives an object as parameter. It is the part of the model corresponding only to the grid which can be passed to the function performing the validation.

The function called by the code block must return a logical value where the row exchange is performed if .T. (true) and not performed if .F. (false).

### 8.3.3 Validation of duplicate row (SetUniqueLine)

In a data model in which grid components are added, the fields that cannot be repeated in this grid are defined.

For example, imagine that we cannot repeat a product code in the Sales Order. We can define this behavior in the model without writing specific function for this.

The method of the data model (Model) that must be used is **SetUniqueLine**.

```
// Liga o controle de não repetição de linha
oModel:GetModel( 'ZA2DETAIL' ):SetUniqueLine( { 'ZA2_AUTOR' } )
```

In the previous example, field **ZA2\_AUTOR** cannot have its content repeated on the grid. More than one field can be informed to create a control with a composed key.

```
oModel:GetModel( 'ZA2DETAIL' ):SetUniqueLine( { 'ZA2_AUTOR', 'ZA2_DATA' } )
```

In the example previous to the combination of field **ZA2\_AUTOR** and **ZA2\_DATA** cannot have its content repeated on the grid.

The repetition can occur individually but not together.

ZA2_AUTOR	ZA2_DATA	
001	01/01/11	Ok
001	02/01/11	Ok
002	02/01/11	Ok
001	01/01/11	Not allowed

### 8.3.4 Pre-validation of row

In a data model in which grid components exist, a validation to be executed in the grid row actions can be defined. By these actions, we can understand the attribution of values, delete or recover a row.

We define the pre-validation of row as a code block in the 4th parameter of the **AddGrid** method.

```
oModel:AddGrid( 'ZA2DETAIL', 'ZA1MASTER', oStruZA2, { |oModelGrid, nLine, cAction, cField| COMP021LPRE(oModelGrid, nLine, cAction, cField) }
```

The code block receives the following as parameter:

- An object that is part of the model corresponding only to the grid;
- The row number;
- The action executed:
  - **SETVALUE** – For attribution of values;
  - **DELETE** – For deletion and recovering of the row.

Field to which the value is attributed, for deletion or recovering of the row, is not passed.

These parameters can be passed for the function performing the validation.

The function called by the code block must return a logical value where the row exchange is performed if .T. (true) and not performed if .F. (false).

An example to use the pre-validation of the row:

```
Static Function COMP023LPRE( oModelGrid, nLinha, cAcao, cCampo )
```

```
Local lRet      := .T.
```

```
Local oModel    := oModelGrid:GetModel()
```

```
Local nOperation := oModel:GetOperation()
```

```
// Valida se pode ou não apagar uma linha do Grid
If cAcao == 'DELETE' .AND. nOperation == MODEL_OPERATION_UPDATE
    lRet := .F.
    Help( ,, 'Help',, 'Não permitido apagar linhas na alteração.' +;
        CRLF + 'Você está na linha ' + Alltrim( Str( nLinha ) ), 1, 0 )
EndIf

Return lRet
```

In the previous example, the deletion of rows in the editing operation is not allowed.

### 8.3.5 Validation of the model activation (*SetVldActivate*)

It is the validation performed in the activation of the model, allowing its activation or not.

We define the validation of the activation using the **SetVldActivate** method.

```
oModel:SetVldActivate( { |oModel| COMP011ACT( oModel ) } )
```

The code block receives an object as parameter. This object belongs to the corresponding model but the model does not have the data loaded yet, as the data load is made after its activation.

The function called by the code block must return a logical value where the activation is performed if .T. (true) and not performed if .F. (false).

## 8.4 Manipulation of the grid component

Now we will learn about some treatments that can be carried out in the grid components of a data model (Model)

### 8.4.1 Amount of grid component rows (*Length*)

To obtain an amount of grid rows, you must use the **Length** method.

The rows deleted are also considered in the counting.

```
Static Function COMP021POS( oModel )
Local lRet      := .T.
Local oModelZA2 := oModel:GetModel( 'ZA2DETAIL' )
Local nI        := 0

For nI := 1 To oModelZA2:Length()
    // Segue a funcao ...
Next nI
```

If a parameter is passed in the **Length** method, the return is only the amount of rows not deleted from the *grid*.



```
nLinhas := oModelZA2.Length( .T. ) // Quantidade linhas não apagadas
```

### 8.4.2 Go to a grid component rows (GoLine)

To transfer the grid, that is, change row which the grid is positioned, we use the **GoLine** method passing the row number which you wish to position as parameter.

```
Static Function COMP021POS( oModel )
Local lRet      := .T.
Local oModelZA2 := oModel:GetModel( 'ZA2DETAIL' )
Local nI        := 0

For nI := 1 To oModelZA2:Length()
    oModelZA2:GoLine( nI )
    // Segue a função ...
Next nI
```

### 8.4.3 Status of the grid component row

When we talk about the data model (*Model*), we have 3 basic operations: **Adding, Editing and Deleting**.

When the operation is inclusion, all the data model (Model) components are added. This also applied to deletion, if this is the operation, all components will have their data deleted.

But when we talk about the editing operation, it is a different thing.

In a data model with grid components, rows can be added to it, edited or deleted in the grid editing operation. That is, the data model (Model) is under editing but a grid can have had 3 operations in its rows.

In *MVC*, you can learn what operations a line suffered by the following status:

**IsDeleted:** Informs is a row was deleted. The line was deleted when it returns .T. (true).

**IsUpdated:** Informs is a row was edited. The line was edited when it returns .T. (true).

**IsInserted:** Informs is a row was added, that is, if it is a new row on the grid. The line was added when it returns .T. (true).

#### **Example:**

```
Static Function COMP23ACAO()

Local oModel      := FWModelActive()
Local oModelZA2    := oModel:GetModel( 'ZA2DETAIL' )
Local nI           := 0
Local nCtInc       := 0
Local nCtAlt       := 0
Local nCtDel       := 0
```

```

Local aSaveLines := FWSaveRows()

For nI := 1 To oModelZA2:Length()
    oModelZA2:GoLine( nI )

    If      oModelZA2:IsDeleted()
        nCtDel++
    ElseIf oModelZA2:IsInserted()
        nCtInc++
    ElseIf oModelZA2:IsUpdated()
        nCtAlt++
    EndIf

Next

Help( ,, 'HELP',, 'Existem na grid' + CRLF + ;
Alltrim( Str( nCtInc ) ) + ' linhas incluídas' + CRLF + ;
Alltrim( Str( nCtAlt ) ) + ' linhas alteradas' + CRLF + ;
Alltrim( Str( nCtDel ) ) + ' linhas apagadas' + CRLF ;
, 1, 0)

```

But a status method can return .T. (true) for the same line. If a line was added, ***IsInserted*** returns .T. (true), then it was edited, ***IsUpdated*** returns .T. (true) and then the same row was deleted, ***IsDeleted*** also returns .T. (true).

#### 8.4.4 Adding a grid row (***AddLine***)

To add a row to the grid component of the data model (Model) we use the ***AddLine*** method.

```

nLinha++
If oModelZA2:AddLine() == nLinha
// Segue a função
EndIf

```

The AddLine method returns the total amount of grid rows. If the *grid* already has 2 rows and everything went well when adding the row, AddLine returns 3. If there was some error, the problem returns 2 as the new row was not inserted.

The reasons why the adding was not successful can be some mandatory field not informed, the post-validation of the row returned .F. (false), reached maximum capacity of rows for the grid, for example.

#### 8.4.5 Deleting and recovering a grid row (***DeleteLine*** and ***UnDeleteLine***)

To delete a row of a grid component of the data model (Model) we use the ***DeleteLine*** method.

```

Local oModel      := FWModelActive()
Local oModelZA2   := oModel:GetModel( 'ZA2DETAIL' )
Local nI          := 0
For nI := 1 To oModelZA2:Length()
    oModelZA2:GoLine( nI )

    If !oModelZA2:IsDeleted()
        oModelZA2>DeleteLine()
    EndIf

Next

```

The ***DeleteLine*** returns .T. (true) if the deletion was successful. A reason it is not is the pre-validation of the row returns .F. (false).

If you wish to recover a deleted grid row, use the ***UnDeleteLine*** method.

```

Local oModel      := FWModelActive()
Local oModelZA2   := oModel:GetModel( 'ZA2DETAIL' )
Local nI          := 0

For nI := 1 To oModelZA2:Length()
    oModelZA2:GoLine( nI )

    If oModelZA2:IsDeleted()
        oModelZA2:UnDeleteLine()
    EndIf

Next

```

The ***UnDeleteLine*** returns .T. (true) if the recovering was successful. A reason it is not is the pre-validation of the row returns .F. (false).

#### **8.4.6 Permissions for a grid**

If you wish to limit a row to be added, edited or deleted, you can query, for example, using the methods below:

**SetNoInsertLine:** Does not allow rows to be added to the grid.

**Example:**

```
oModel:GetModel( 'ZA2DETAIL' ):SetNoInsertLine( .T. )
```

**SetNoUpdateLine:** Does not allow grid rows to be edited.

**Example:**

```
oModel:GetModel( 'ZA2DETAIL' ):SetNoUpdateLine( .T. )
```

**SetNoDeleteLine:** Does not allow grid rows to be deleted.

**Example:**

```
oModel:GetModel( 'ZA2DETAIL' ):SetNoDeleteLine( .T. )
```

These methods can be entered when defining the data model (Model).

#### **8.4.7 Permission of grid with no data (SetOptional)**

By standard, when we have a data model (*Model*) with a grid component, you must enter at least a row in this grid.

Imagine a model with the registration of products and accessories. It is a Master-Detail model. We have *n* accessories for each product, but also products with no accessories. So the rule that requires at least one row entered in the grid cannot be applied.

In this case, we use the **SetOptional** method to allow the grid to have at least one row entered or not, that is entering the grid data is optional.

This method must be informed after defining the data model (Model).

**Example:**

```
oModel:GetModel( 'ZA2DETAIL' ):SetOptional( .T. )
```

If a grid is optional and there are mandatory fields in the structure, it is only validated if these fields are informed and if the row goes through some editing of any field.

The **IsOptional** method can be used to know if the grid component has this feature or not. If it returns .T. (true), the component allows typed rows not to exist. This method can be useful in validations.

#### **8.4.8 Saving and restoring the grid position (FWSaveRows / FWRestRows)**

A care you must have when writing a function, even if not for MVC use, is to restore the areas of the tables not positioned anymore.

Similarly, you should have the same care for the grid components of are not positioned anymore in a function, using the **GoLine** method.

For this, use the **FWSaveRows** functions to save the data model (Model) grid row position and

**FWRestRows** to restore these positions.

**Example:**

```
Static Function COMP23ACAO()  
  
Local oModel      := FWModelActive()  
Local oModelZA2    := oModel:GetModel( 'ZA2DETAIL' )  
Local nI           := 0  
Local aSaveLines   := FWSaveRows()  
  
For nI := 1 To oModelZA2:Length()  
    oModelZA2:GoLine( nI )  
  
    // Segue a função  
Next  
  
FWRestRows( aSaveLine )
```

**Note:** **FWSaveRows** saves the position of all data model 9Model) grids and **FWSaveRows** restores the positions of all model grids.

#### **8.4.9 Definition of maximum amount of grid rows (SetMaxLine)**

By standard, the maximum amount of rows of a grid component is 990.

If necessary, edit this amount using the **SetMaxLine** method. This method must be used in the definition of the data model (Model), that is, **ModelDef**.

**Important:** The amount always regards the total of rows, whether they are deleted or not.

### **8.5 Obtaining and attributing values to the data model**

The most common operations we will do in a data model 9Model) is obtaining and attributing values.

For this, use one of the methods below:

**GetValue:** Obtain a data of the data model (Model). We can obtain the data from the complete model or from its component.

From the complete data model (Model).

```
Local cMusica := oModel:GetValue( 'ZA1MASTER', 'ZA1_MUSICA' )
```

In which **ZA1MASTER** is the component identifier (ID) and **ZA1\_MUSICA** is the field from which you wish to obtain the data.

Or from a data model (Model) component.

```
Local oModelZA2 := oModel:GetModel( 'ZA1MASTER' )
Local cMusica := oModelZA2:GetValue('ZA1_MUSICA' )
```

**SetValue:** Attributes a data to the data model (Model). We can attribute the data from the complete model or from a part of it.

From the complete data model (Model).

```
oModel:SetValue( 'ZA1MASTER', 'ZA1_MUSICA', '000001' )
```

In which **ZA1MASTER** is the component identifier (ID) and **ZA1\_MUSICA** is the field in which you wish to attribute the data and **000001** is the data you wish to attribute.

Or from a data model (Model) component.

```
Local oModelZA2 := oModel:GetModel( 'ZA1MASTER' )
oModelZA2:SetValue('ZA1_MUSICA', '000001' )
```

When we use **SetValue** to attribute a data to a field, the validations of this field are executed and triggered.

**SetValue** returns .T. (true) if the attribution succeeds, the reasons not to can be the data not satisfying the validation or the editing mode (**WHEN**) was not satisfied.

**LoadValue:** Attributes a data to the data model (Model). We can attribute the data from the complete model or from a part of it.

From the complete data model (Model).

```
oModel:LoadValue( 'ZA1MASTER', 'ZA1_MUSICA', '000001' )
```

In which **ZA1MASTER** is the component identifier (ID) and **ZA1\_MUSICA** is the field in which you wish to attribute the data and **000001** is the data you wish to attribute.

Or from a data model (Model) component.

```
Local oModelZA2 := oModel:GetModel( 'ZA1MASTER' )
...
oModelZA2:LoadValue('ZA1_MUSICA', '000001' )
```

The difference between **LoadValue** and **SetValue** is that **LoadValue** does not validate nor trigger the field. It forces the data attribution.

**Important:** Always use **SetValue** to attribute a data, avoid **LoadValue**. Only use it when extremely necessary.

## 8.6 Behavior

We will learn how to edit some standard data model (Model) behaviors.

### 8.6.1 Editing component data in the data model (SetOnlyView)

If we want a certain data model (Model) component not to edit its data and be used only for view, use the **SetOnlyView** method.

This method must be entered when defining the Model.

#### Example:

```
oModel:GetModel( 'ZA2DETAIL' ):SetOnlyView ( .T. )
```

### 8.6.2 Not saving data of a component of the data model (SetOnlyQuery)

The persistence of the data (saving) is made automatically by the data model (Model).

If we want a certain data model (Model) component to allow the adding and/or editing its data, but not saving this data, use the **SetOnlyView** method.

This method must be entered when defining the Model.

#### Example:

```
oModel:GetModel( 'ZA2DETAIL' ):SetOnlyQuery ( .T. )
```

### 8.6.3 Obtaining the operation in execution (GetOperation)

To know the operation with which a data model (Model) is working, we use the **GetOperation** method.

This method returns:

- Value 3 is an **inclusion**;
- Value 4 is a **modification**;
- Value 5 is a **deletion**;

```
Static Function COMP023LPRE( oModelGrid, nLinha, cAcao, cCampo )
```

```
Local lRet      := .T.
```

```
Local oModel    := oModelGrid:GetModel()
```

```
Local nOperation := oModel:GetOperation()
```

```
// Valida se pode ou não apagar uma linha do Grid
```

```
If cAcao == 'DELETE' .AND. nOperation == 3
```

```
    lRet := .F.
```

```
    Help( ,, 'Help',, 'Não permitido apagar linhas na alteração.' + CRLF + ;
```

```
    'Você esta na linha ' + Alltrim( Str( nLinha ) ), 1, 0 )
```



```
EndIf
```

```
Return lRet
```

Several compilation policies **#DEFINE** were created in MVC to make the development easier as well as the reading of an application.

To use this **#DEFINE** you must add the following policy in the source:

```
#  
INCLUDE 'FWMVCDEF.CH'
```

For the operations of the data model (Model), you can use:

- **MODEL\_OPERATION\_INSERT** to **add**;
- **MODEL\_OPERATION\_UPDATE** to **edit**;
- **MODEL\_OPERATION\_DELETE** to **delete**.

In the example given, we can enter:

```
If cAcao == 'DELETE' .AND. nOperation == MODEL_OPERATION_INSERT
```

#### 8.6.4 Data manual saving (FWFormCommit)

The saving (persistence) data model (Model) data is carried out by the MVC in which all the model entities data are saved.

However, you may have the need to save in other entities that are not a part of the model. For example, when we add a Sales Order, you must update the pending value of orders in the Customers File. The header and items of the order are part of the model and will be saved. The Customer file is not a part but needs to be saved as well.

For this type of situation, you can intervene when saving the data.

So we defined a code block in the 4th parameter of construction class of the **MPFormModel** **data** model (Model).

```
oModel := MPFormModel():New( 'COMP011M', , , { |oModel| COMP011GRV( oModel ) } )
```

The code block receives an object as parameter. It is the model which can be passed to the function performing the saving.

Unlike code blocks defined in the data model (Model) to validate and complement validations made by MVC, the code block for saving replaces data saving. So after defining a code block for saving, it becomes a responsibility of the created function, the saving of all data including the data model data in use.

To make the development easier, **FWFormCommit** function was created to save the data of the informed data model (Model) objects.

```
Static Function COMP011GRV ( oModel )
```

```
FWFormCommit( oModel )
```

```
// Efetuar a gravação de outros dados em entidade que  
// não são do model
```

**Important:** No attributions must be made to the data model (Model) within the saving function. Conceptually when you start saving, the data model (Model) has already been through all the validation. When you try to assign a value, this value may not meet the validation of the field making the data model (Model) invalidated again and this will occur and inconsistent data will be saved.

## 8.7 Completion rules (*AddRules*)

New feature implemented on MVC is the completion rules, in which the completion of a field depends on the completion of another.

For example, we can establish that the Unit Code field of an entity can only be filled out after the completion of the Customer Code field.

The completion rules can be of 3 types:

- **Type 1 Pre-Validation:**

It adds a relationship of dependence among the form fields, preventing the attribution of values if the dependence fields have no value assigned. For example, the completion of the Unit Code field can only be filled out after filling out the Customer Code field.

- **Type 2 Post-Validation**

It enables a relationship of dependence between the origin and destination reference, causing a reevaluation of destination in the event of origin update. For example, after completing the Unit Code field, the validation is reevaluated if the Customer Code is edited.

- **Type 3 Pre and Post-Validation**

Types 1 and 2 simultaneously

### **Example:**

```
oModel:AddRules( 'ZA3MASTER', 'ZA3_LOJA', 'ZA3MASTER', 'ZA3_DATA', 1 )
```

**ZA3MASTER** is the identifier (*ID*) of the data model (Model) component where the destination field is, **ZA3\_LOJA** is the destination field, the second **ZA3MASTER** is the data model (Model) component where the origin field is, **ZA3\_DATA** is the origin field.

## 9.Interface treatments

We will see some treatments that can be carried out in the interface (View) according to need:

- Creating buttons;
- Creating folders;

- Grouping fields;
- Incrementing fields;
- Etc.

## 9.1 Incremental field (*AddIncrementField*)

We can make a data model (Model) field be a part of a grid component so that it can be unitarily incremented with each new row inserted.

For example, imagine the Sales Order in the items, the item number can be an incremental field.

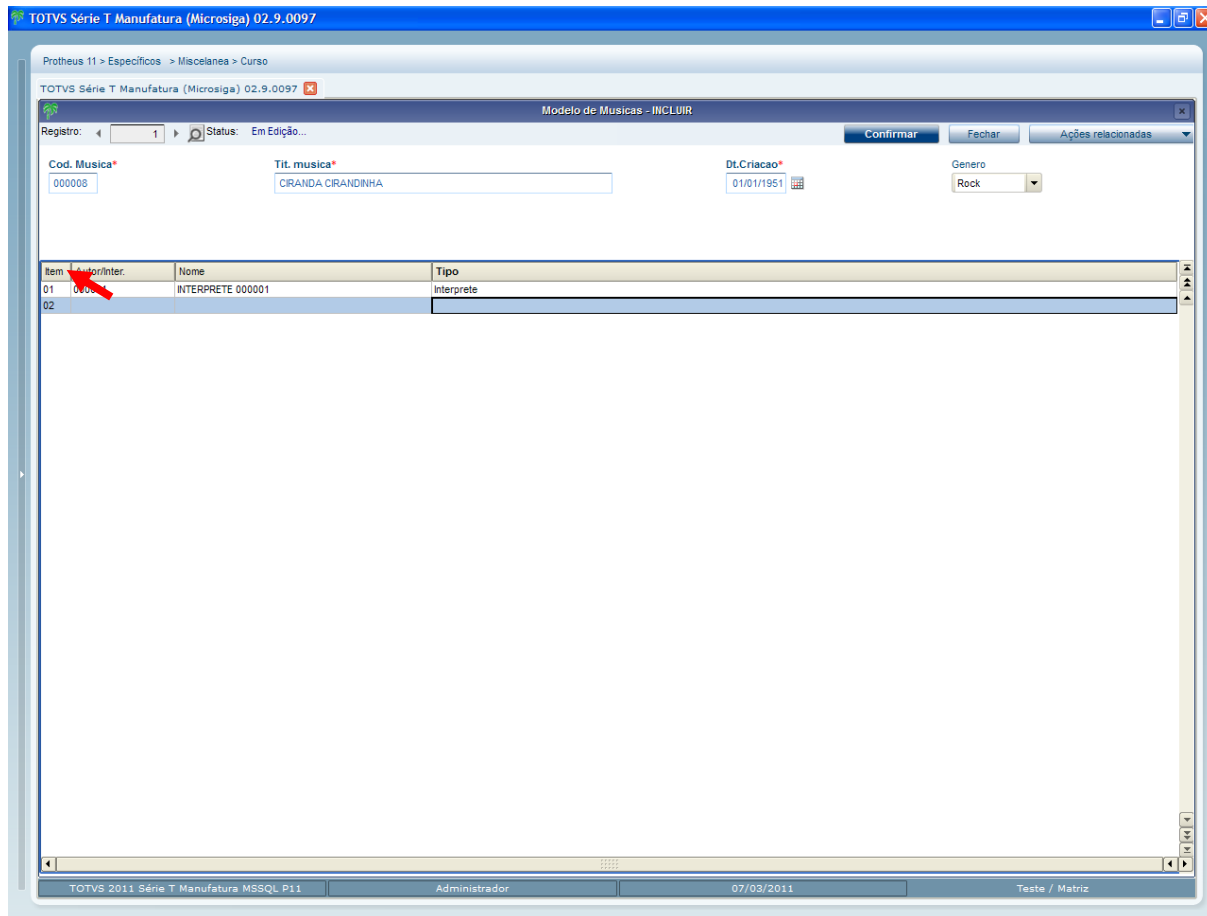
For this, we use the **AddIncrementField** method.

### **Example:**

```
oView.AddIncrementField( 'VIEW_ZA2', 'ZA2_ITEM' )
```

Where **VIEW\_ZA2** is the identifier (ID) of the interface (View) component, in which the field is **ZA2\_ITEM**, the field name is incremented.

Visually we have:



**Important:** This behavior only happens when the application is being used by its interface (View). When the data model is used directly (Web Services, automatic routines, etc.) the incremental field must be informed normally.

## 9.2 Creation of buttons in the buttons bar (AddUserButton)

To create buttons in the interface buttons bar, we use the **AddUserButton** method.

### Example:

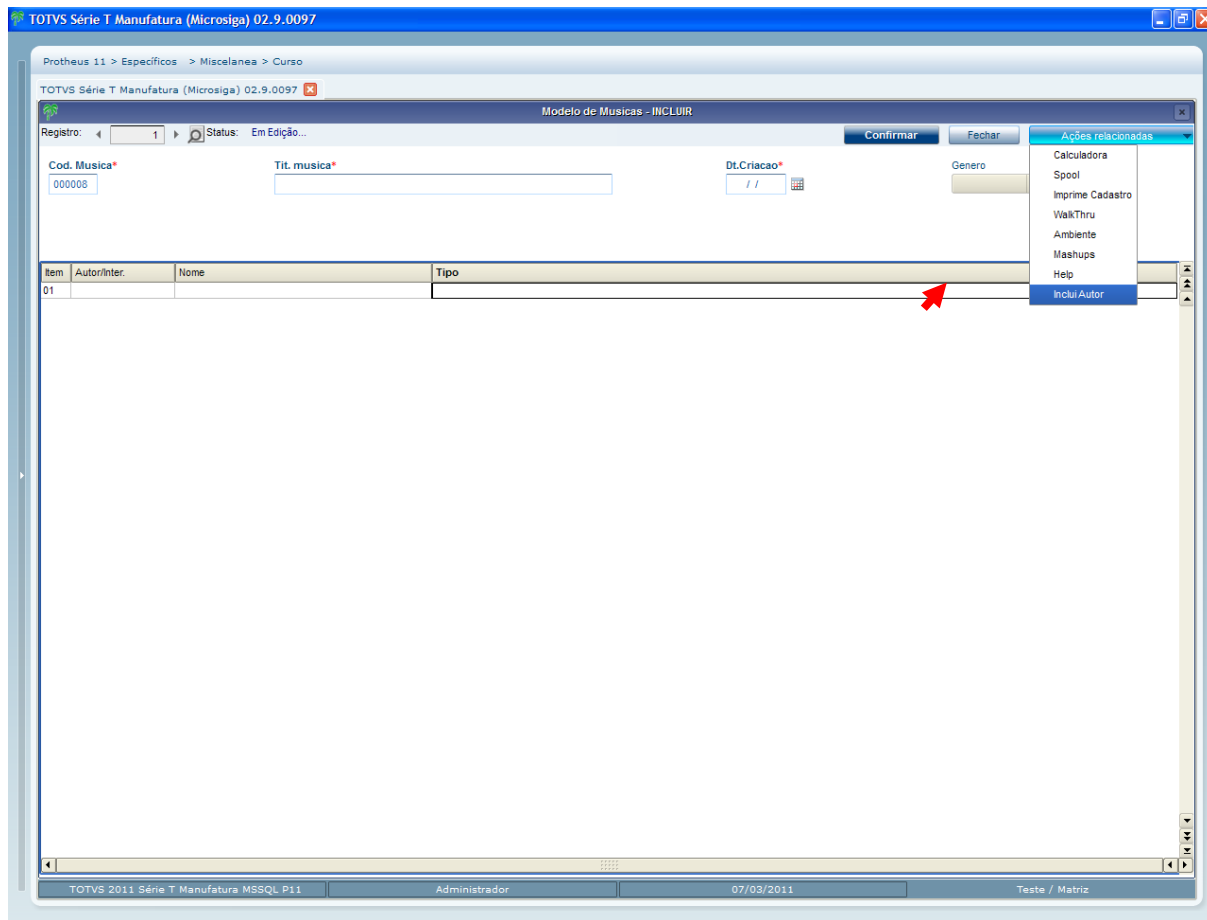
```
oView:AddUserButton( 'Inclui Autor', 'CLIPS', {|oView| COMP021BUT()})
```

Where the **Add Author** is the text to be displayed in the button, **CLIPS** is the name of the RPO<sup>2</sup> image to be used for the button and the 3rd parameter is the code block to be executed when adding a button.

---

<sup>2</sup> RPO – Microsiga Protheus Repository for applications and images

Visually we have:



### 9.3 Component title (*EnableTitleView*)

In *MVC* we can assign a title to identify each interface component, so we use the ***EnableTitleView*** method.

#### ***Example:***

```
oView.EnableTitleView('VIEW_ZA2', 'Musicas')
```

Where ***VIEW\_ZA2*** is the identifier of the *interface (View)* component, and ***'MUSIC'*** is the desired title for the component

We can also use:

```
oView.EnableTitleView('VIEW_ZA2')
```

Where the title to be displayed is what was defined in the ***SetDescription*** method of the data model (Model) for the component.

Visually we have:

TOTVS Série T Manufatura (Microsiga) 02.9.0097

Protheus 11 > Especificos > Miscelanea > Curso

TOTVS Série T Manufatura (Microsiga) 02.9.0097

Modelo de Musicas - VISUALIZAR

Registro: 1 Status: Fechar Ações relacionadas

Cod. musica\* 000000 Tit. musica\* TESTE GHGJ Dt.Criacao\* 28/06/2010 Genero Blues

**Musicas**

Item	Autor/Inter.	Nome	Tipo
02	000002	INTERPRETE 000002	2
03	000001	INTERPRETE 000001	2
05	000003	INTERPRETE 000003	2
06	000004	AUTOR 000004	1
07	000005	INTERPRETE 000005	2
08	000006	AUTOR 000006	1
09	000009	INTERPRETE 000009	2

TOTVS 2011 Série T Manufatura MSSQL P11 Administrador 10/03/2011 Teste / Matriz

## 9.4 Editing the Fields in grid components (SetViewProperty)

A new MVC feature when using the interface is for a grid component to edit the the data directly on the grid and/or on a screen in the form layout.

For this, we use the **SetViewProperty** method. This method enables some specific behaviors to the *interface* (*View*) component, according to the received policy.

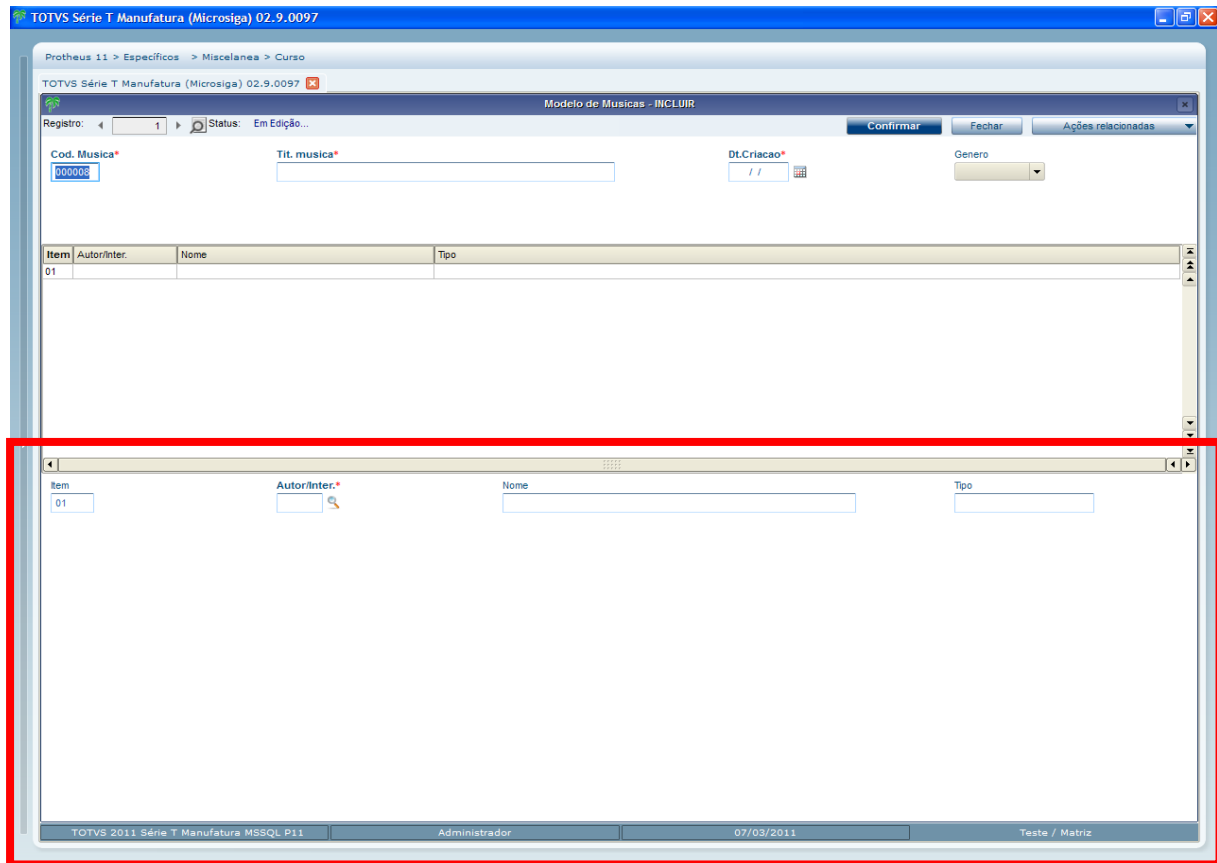
### Example:

```
oView.SetViewProperty( 'VIEW_ZA2', "ENABLEDGRIDDETAIL", { 60 } )
```

Where **VIEW\_ZA2** is the identifier (ID) of the *interface* (*View*) component, and the **ENABLEDGRIDDETAIL** is the policy that enables the behavior

**{60}** is the percentage that the editing form takes from the size that the grid component currently takes. Exemplifying numerically, if it was defined in the grid component that it will use 50% of the screen. When working with 60 (60%) in the parameter, 50% is destined to the grid component, 60% is used for the editing form.

Visually we have:



## 9.5 Creating folders (CreateFolder)

In *MVC* we can create folders where interface (View) components are included.

For this, we use the **CreateFolder** method.

### Example:

```
oView.CreateFolder( 'PASTAS' )
```

We must give an identifier (ID) for each interface (View) component. **FOLDERS** is the identifier (*ID*) given to the folders.

After the creation of the main folder, you must create the tabs of this folder. For this, we use the **AddSheet** method.

For example, we will create 2 tabs:

```
oView.AddSheet( 'PASTAS', 'ABA01', 'Cabeçalho' )
oView.AddSheet( 'PASTAS', 'ABA02', 'Item' )
```

Where **FOLDERS** is the identifier (ID) of the folder and **TAB01** and **TAB02** are the IDs given to each tab and the **Header** and **Item** are the titles of each tab.

We must always create a container, an object, to receive some interface (View) element.

A forma para se criar um **box** em uma aba é:

```
oView.CreateHorizontalBox( 'SUPERIOR', 100,,, 'PASTAS', 'ABA01' )
```

```
oView.CreateHorizontalBox( 'INFERIOR', 100,,, 'PASTAS', 'ABA02' )
```

We must give an identifier (ID) for each interface (View) component.

- **SUPERIOR** and **INFERIOR** are the IDs of each **box**.
- **100** indicates the percentage that the **box** will take from the tab.
- **FOLDERS** is the identifier (*ID*) of the folder.
- **ABA01** and **ABA02** are the IDs of the tabs.

We have to relate the interface (View) component with a **box** to display, so we use the **SetOwnerView** method.

```
oView.SetOwnerView( 'VIEW_ZA1' , 'SUPERIOR' )
```

```
oView.SetOwnerView( 'VIEW_ZA2' , 'INFERIOR' )
```

Summing up:

```
// Cria Folder na view
```

```
oView.CreateFolder( 'PASTAS' )
```

```
// Cria pastas nas folders
```

```
oView.AddSheet( 'PASTAS', 'ABA01', 'Cabeçalho' )
```

```
oView.AddSheet( 'PASTAS', 'ABA02', 'Item' )
```

```
// Criar "box" horizontal para receber algum elemento da view
```

```
oView.CreateHorizontalBox( 'GERAL' , 100,,, 'SUPERIOR', 'ABA01' )
```

```
oView.CreateHorizontalBox( 'CORPO' , 100,,, 'INFERIOR', 'ABA02' )
```

```
// Relaciona o identificador (ID) da View com o "box" para exibição
```

```
oView.SetOwnerView( 'VIEW_ZA1' , 'SUPERIOR' )
```

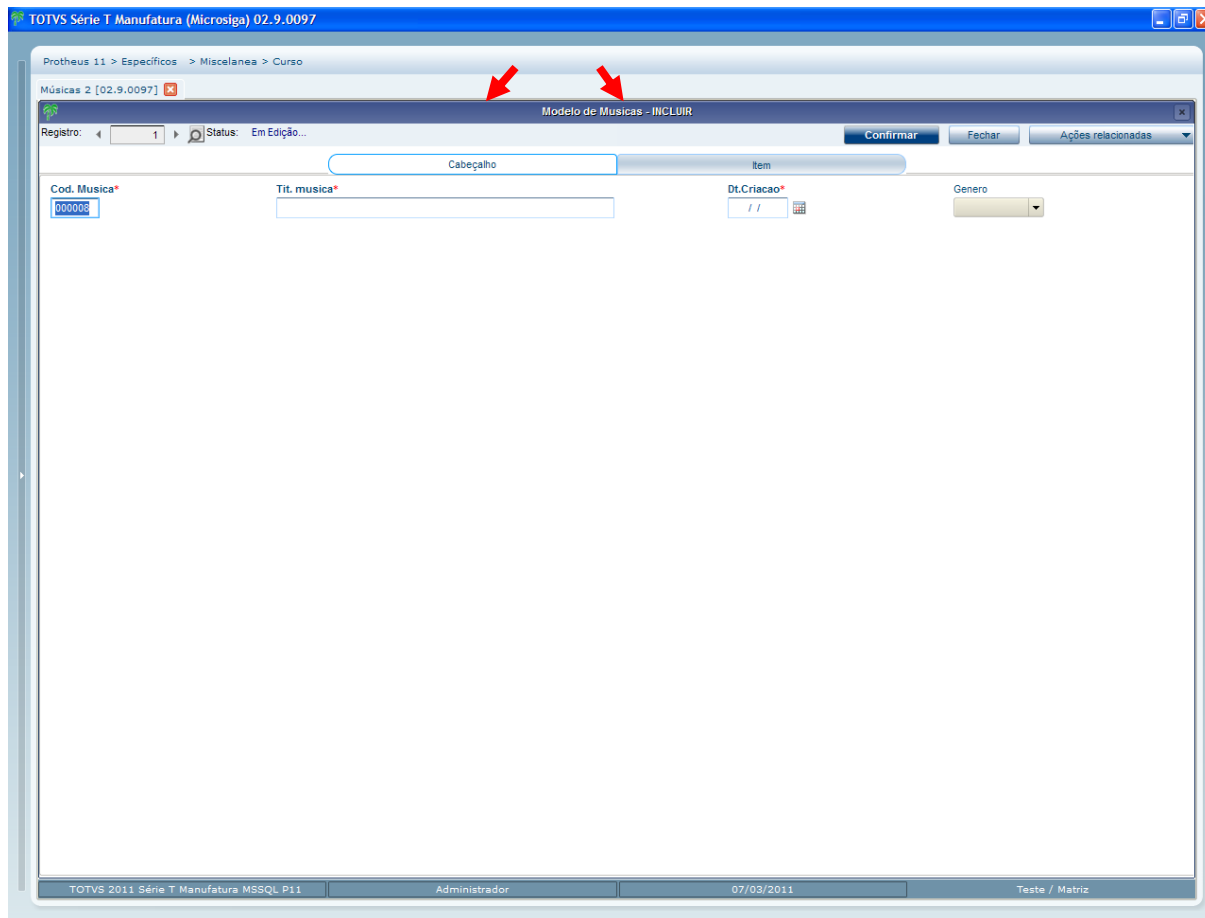
```
oView.SetOwnerView( 'VIEW_ZA2' , 'INFERIOR' )
```

When the folders are established using metadata (dictionaries), the interface (View)



automatically creates these folders. If the component added in one of the tabs has folders established in the metadata, these folders are created in the tab where the component found.

Visually we have:



## 9.6 Grouping fields (AddGroup)

One new feature that MVC has for the interface use is for a form component to group fields in the screen.

For example, in a customers files we may have fields for delivery, correspondence and billing. For a better view, we can group fields of each address.

For this, we use the **AddGroup** method.

### Example:

```
oStruZA0:AddGroup( 'GRUPO01', 'Alguns Dados', '', 1 )
oStruZA0:AddGroup( 'GRUPO02', 'Outros Dados', '', 2 )
```

We must give an identifier (ID) for each interface (View) component.

**GRUPO01** is the identifier (ID) given to the grouping, the 2nd parameter is the title to be displayed in the grouping and 1 is the type of grouping. It can be 1 - Window, 2 - Separator

With the grouping created, we must say what fields are part of this grouping. For this, we edit a property of the structure of some fields. We will use the **SetProperty** method which can be seen with more details in chapter [Erro! Fonte de referência não encontrada](#). [Erro! Fonte de referência não encontrada](#).

```
// Colocando todos os campos para um agrupamento'  
oStruZA0:SetProperty( '*' , MVC_VIEW_GROUP_NUMBER, 'GRUPO01' )  
// Trocando o agrupamento de alguns campos  
oStruZA0:SetProperty( 'ZA0_QTDMUS', MVC_VIEW_GROUP_NUMBER, 'GRUPO02' )  
oStruZA0:SetProperty( 'ZA0_TIPO' , MVC_VIEW_GROUP_NUMBER, 'GRUPO02' )
```

Visually we have:

**Note:** Groupings are displayed in the *interface (View)* in the order they were created.

## 9.7 Interface action (*SetViewAction*)

You can execute a function in MVC in some of the interface (View) actions. This resource can be used when we want to run something in the interface and not have reflex in the data model (Model) such as a **Refresh** of the screen, for example.

This is possible in the following actions:

- Interface refresh;
- Enabling the button to confirm the interface;
- Enabling the button to cancel the interface;
- Deleting the grid row;
- Restoring the grid row;

For this, we use the **SetViewAction** method.

The syntax is:

```
oView:SetViewAction( <cActionlID>, <bAction> )
```

Where:

**cActionlID** ID if the point in which the action is performed, which can be:

<b>REFRESH</b>	Performs the action in the View's <b>Refresh</b> ;
<b>BUTTONOK</b>	Performs the action when enabling the View's Confirm button;
<b>BUTTONCANCEL</b>	Performs the action when enabling the View's Cancel button;
<b>DELETELINE</b>	Performs the action when deleting the grid row;
<b>UNDELETELINE</b>	Performs the action when restoring the grid line;

**bAction** Block with the action to be performed. Receives as parameter:

<b>REFRESH</b>	Receives the View object as parameter;
<b>BUTTONOK</b>	Receives the the View object as parameter;
<b>BUTTONCANCEL</b>	Receives the the View object as parameter;
<b>DELETELINE</b>	Receives the View object, View identifier (ID) and row number as parameter.
<b>DELETELINE</b>	Receives the View object, View identifier (ID) and row number as parameter.

### **Example:**

```
oView:SetViewAction( 'BUTTONOK'      ,{ |oView| SuaFuncao( oView ) } )
oView:SetViewAction( 'BUTTONCANCEL',{ |oView| OutraFuncao( oView ) } )
```

**Important:** These actions are run only when an interface (View) exists. It does not occur when we have the direct instancing of the model, automatic routine or Web Service. Avoid putting actions in these functions that may influence the business rules as the application execution without interface are not executed.

## **9.8 Interface action of the field (SetFieldAction)**

You can execute a function in MVC after the validation of the field of some data model (Model) component.

This resource can be used when we wish to run something in the interface with no reflex on the model, such as a screen **Refresh** or opening an auxiliary screen, for example.

For this, we use the **SetFieldAction** method.

The syntax is:

```
oView:SetFieldAction( <cIDField>, <bAction> )
```

Where:

**cIDField**      Field (name) ID:

**bAction**      Block with an action executed, receives as parameter:

- View Object
- The View identifier (ID).
- The Field identifier (ID).
- Field Content

### **Example:**

```
oView:SetFieldAction( 'A1_COD', { |oView, cIDView, cField, xValue| SuaFuncao( oView, cIDView, cField, xValue ) } )
```

### **Important:**

- These actions are performed after the field validation.
- These actions are run only when an interface (View) exists. It does not occur when we have the direct instancing of the model, automatic routine or Web Service.
- Avoid putting actions in these functions that may influence the business rules as the application execution without interface are not executed.

## **9.9 Other objects (AddOtherObjects)**

In the construction of some applications we may have to add a component that is not part of the MVC standard interface, such a chart, a calendar, etc. to the interface.

For this, we use the **AddOtherObject** method.

The syntax is:

**AddOtherObject**( <Id>, <Code Block to be executed> )

Where the 1st parameter is the identifier (*ID*) do **AddOtherObjects** and the 2nd parameter is the block code to be executed for creation **of other objects**.

MVC is limited to making the function call. The responsibility to construct and update the data is up to the function developer.

### Example:

```
AddOtherObject( "OTHER_PANEL", { |oPanel| COMP23BUT( oPanel ) } )
```

Notice that the 2nd parameter receives an object which is the **container** where the developer must place objects.

Below is an example of the method use, where we place two buttons of in interface parts. Notice the comments on the source:

```
oView := FWFormView():New()
oView:SetModel( oModel )

oView:AddField( 'VIEW_ZA3', oStruZA3, 'ZA3MASTER' )
oView:AddGrid( 'VIEW_ZA4', oStruZA4, 'ZA4DETAIL' )
oView:AddGrid( 'VIEW_ZA5', oStruZA5, 'ZA5DETAIL' )

// Criar "box" horizontal para receber algum elemento da view
oView:CreateHorizontalBox( 'EMCIMA' , 20 )
oView:CreateHorizontalBox( 'MEIO' , 40 )
oView:CreateHorizontalBox( 'EMBAIXO', 40 )

// Quebra em 2 "box" vertical para receber algum elemento da view
oView:CreateVerticalBox( 'EMBAIXOESQ', 20, 'EMBAIXO' )
oView:CreateVerticalBox( 'EMBAIXODIR', 80, 'EMBAIXO' )

// Relaciona o identificador (ID) da View com o "box" para exibicao
oView:SetOwnerView( 'VIEW_ZA3', 'EMCIMA' )
oView:SetOwnerView( 'VIEW_ZA4', 'MEIO' )
oView:SetOwnerView( 'VIEW_ZA5', 'EMBAIXOESQ' )
// Liga a identificacao do componente
oView:EnableTitleView( 'VIEW_ZA3' )
oView:EnableTitleView( 'VIEW_ZA4', "MÚSICAS DO ÁLBUM" )
oView:EnableTitleView( 'VIEW_ZA5', "INTERPRETES DAS MÚSICAS" )
```

```

// Acrescenta um objeto externo ao View do MVC
// AddOtherObject(cFormModelID,bBloco)
// cIDObject - Id

// bBloco      - Bloco chamado devera ser usado para se criar os objetos de tela externos
ao MVC.

oView:AddOtherObject("OTHER_PANEL", {|oPanel| COMP23BUT(oPanel)})

// Associa ao box que ira exibir os outros objetos
oView:SetOwnerView("OTHER_PANEL",'EMBAIXODIR')

Return oView

//-----
Static Function COMP23BUT( oPanel )
Local lOk := .F.

// Ancoramos os objetos no oPanel passado
@ 10, 10 Button 'Estatistica'   Size 36, 13 Message 'Contagem da FormGrid' Pixel Action
COMP23ACAO( 'ZA4DETAIL', 'Existem na Grid de Musicas' ) of oPanel

@ 30, 10 Button 'Autor/Inter.'   Size 36, 13 Message 'Inclui Autor/Interprete' Pixel
Action FWExecView('Inclusao por FWExecView','COMP011_MVC', MODEL_OPERATION_INSERT, , { ||
.T. } ) of oPanel

Return NIL

```

Visually we have:

TOTVS Série T Serviços (Microsiga) 02.9.0097

Protheus 11 > Específicos > Miscelanea > Curso

TOTVS Série T Serviços (Microsiga) 02.9.0098

Modelo de Albums - INCLUIR

**Dados do Album**

Cod. album\* 000004 Descrição Data\* / /

**MÚSICAS DO ÁLBUM**

Musica	Tit. musica
--------	-------------

**INTERPRETES DAS MÚSICAS**

Interprete	Nome
------------	------

Registro: 1 Status: Em Edição...

Confirmar Fechar Ações relacionadas

TOTVS 2011 Série T Serviços MSSQL P11 Administrador 04/05/2011 Teste / Matriz

## 10.Treatment of data structure

As we previously said, MVC does not work linked to Microsiga Protheus metadata. It works linked to structures. These structures can be constructed from the metadata (dictionaries).

We will see some treatments that can be carried out in the structures according to need:

### 10.1 Selection of fields for structure (*FWFormStruct*)

When we create a structure based on the metadata (dictionaries), using the **FWFormStruct** function, it takes all the entity fields into account, respecting level, module, use, etc.

If we wish to select what metadata fields (dictionaries) are part of the structure, we must use the 3rd parameter of **FWFormStruct**, which is a block code to be run for each field that the function brings from the metadata (dictionaries) and that receives the field name as parameter.

The code block must return a logical value, in which the field is part of the structure if .T. (true), and not part of the structure if .F. (false).

#### **Example:**

```
Local oStruZA0 := FWFormStruct( 2, 'ZA0', { |cCampo| COMP11STRU(cCampo) } )
```

Where the function can be:

```
Static Function COMP11STRU( cCampo )
```

```
Local lRet := .T.
```

```
If cCampo == 'ZA0_QTDMUS'
```

```
    lRet := .F.
```

```
EndIf
```

```
Return lRet
```

In the function example given, the field **ZA0\_QTDMUS** is not a part of the structure.

The fields dictionary (SX3) of the metadata is positioned for each field.

**Important:** This treatment can be made for the structures to be used in the data model (Model) and the interface (View).

**Be careful:** If a mandatory field is removed from the interface structure (View), even if it is not used to be displayed to the user, the data model (Model) performs its validation by showing that a mandatory field was not mandatory.



## 10.2 Removing structure fields (*RemoveField*)

A way to remove a field from the structure is to use the **RemoveField** method.

### **Example:**

```
Local oStruZA0 := FWFormStruct( 2, 'ZA0')
```

```
oStruZA0: RemoveField('ZA0_QTDMUS')
```

In the function example given, the field **ZA0\_QTDMUS** is not a part of the structure.

**Important:** This treatment can be made for the structures to be used in the data model (Model) and the interface (View).

**Be careful:** If a mandatory field is removed from the interface structure (View), even if it is not used to be displayed to the user, the data model (Model) performs its validation by showing that a mandatory field was not mandatory.

## 10.3 Editing field properties (*SetProperty*)

When we create a structure based on metadata (dictionaries) by using the **FWFormStruct** function the properties that the field has as validation, default started and editing mode are respected.

If you need to edit the need to change some field property in the structure, use the **SetProperty** method.

```
oStruZA0:SetProperty( 'ZA0_QTDMUS' , MODEL_FIELD_WHEN, 'INCLUI')
```

Where the 1st parameter is the name of the field you wish to edit or assign a property, the 2nd is the property regarded and the 3rd is the value for the property.

In the previous example, the field **ZA0\_QTDMUS** can only be edited in the inclusion operation.

The properties for the structure fields in the data model (Model) are:

<b>Properties for the structure fields in the data model (Model):</b>	<b>Type</b>	<b>Description</b>
MODEL_FIELD_TITULO	C	Title
MODEL_FIELD_TOOLTIP	C	Complete Description of the Field
MODEL_FIELD_IDFIELD	C	Name (ID)
MODEL_FIELD_TIPO	C	Type
MODEL_FIELD_TAMANHO	N	Size
MODEL_FIELD_DECIMAL	N	Decimals
MODEL_FIELD_VALID	b	Validation
MODEL_FIELD_WHEN	B	Editing mode
MODEL_FIELD_VALUES	A	List of the field values allowed (combo)
MODEL_FIELD_OBRIGAT	L	Indicates if the completion of the field is mandatory
MODEL_FIELD_INIT	B	Default starter
MODEL_FIELD_KEY	L	Indicates whether the field is key.
MODEL_FIELD_NOUPD	L	Indicates whether the field can receive a value in an update.
MODEL_FIELD_VIRTUAL	L	Indicates whether the field is virtual.

The properties for the structure fields in the data interface (View) are:

Properties for the structure fields in the data interface (View):	Type	Description
MVC_VIEW_IDFIELD	C	Field Name
MVC_VIEW_ORDEM	C	Order
MVC_VIEW_TITULO	C	Field title:
MVC_VIEW_DESCR	C	Field Description
MVC_VIEW_HELP	A	Array with Help
MVC_VIEW_PICT	C	Picture
MVC_VIEW_PVAR	B	Block of Picture Var
MVC_VIEW_LOOKUP	C	Query F3
MVC_VIEW_CANCHANGE	L	Indicates whether the field is editable.
MVC_VIEW_FOLDER_NUMBER	C	Field folder
MVC_VIEW_GROUP_NUMBER	C	Field grouping:
MVC_VIEW_COMBOBOX	A	List of field values allowed (Combo)
MVC_VIEW_MAXTAMCMB	N	Maximum size of the largest combo option
MVC_VIEW_INIBROW	C	Browse starter
MVC_VIEW_VIRTUAL	L	Indicates whether the field is virtual.
MVC_VIEW_PICTVAR	C	Variable Picture

The names of properties mentioned in the data model are actually compilation policies of the **#DEFINE** type.

To use this **#DEFINE** you must add the following policy in the source:

```
#INCLUDE 'FWMVCDEF.CH'
```

You can also assign a property to all structure fields at once by using the name of the asterisk field "\*"

```
oStruZA0:SetProperty( '*' , MODEL_FIELD_WHEN, 'INCLUI')
```

## 10.4 Creation of additional fields in the structure (*AddField*)

If we wish to create a field in an already existing structure, we use the **Addfield** method.

There are differences in the sequence of parameters of this method to add fields to the data model (Model) or interface (View) structure.

The syntax for the data model (Model) is:

**AddField** (cTitulo, cTooltip, cIdField, cTipo, nTamanho, nDecimal, bValid, bWhen, aValues, IObrigat, bInit, IKey, INoUpd, IVirtual, cValid)

### Where:

**cTitulo** Field Title;

**cTooltip** Field Tooltip;

**cIDField** Field ID;

**cTipo** Field Type;

**nTamanho** Field size;

**nDecimal** Field decimal;

**bValid** Field validation code-block;

**bWhen** Field editing mode validation code-block;

**aValues** List of field values allowed;

**IObrigat** Indicates if the completion of the field is mandatory;

**bInit** Field starter code-block;

**IKey** Indicates whether the field is key.

**INoUpd** Indicates whether the field cannot receive value in an update operation;

**IVirtual** Indicates whether the field is virtual.

We exemplified the use below:

```
Local oStruZA0 := FWFormStruct( 1, 'ZA0' )

oStruZA0:AddField( ; // Ord. Tipo Desc.
AllTrim( 'Exemplo 1' ) , ; // [01] C Titulo do campo
AllTrim( 'Campo Exemplo 1' ) , ; // [02] C Tooltip do campo
'ZA0_XEXEM1' , ; // [03] C identificador (ID) do Field
'C' , ; // [04] C Tipo do campo
1 , ; // [05] N Tamanho do campo
0 , ; // [06] N Decimal do campo

FwBuildFeature( STRUCT_FEATURE_VALID, "Pertence('12')"), ; // [07] B Code-block de
validação do campo

NIL , ; // [08] B Code-block de validação When do
campo

{'1=Sim', '2=Não'} , ; // [09] A Lista de valores permitido do
campo

NIL , ; // [10] L Indica se o campo tem
preenchimento obrigatório

FwBuildFeature( STRUCT_FEATURE_INIPAD, "'2'" ) , ; // [11] B Code-block de
inicializacao do campo

NIL , ; // [12] L Indica se trata de um campo chave

NIL , ; // [13] L Indica se o campo pode receber
valor em uma operação de update.

.T. ) // [14] L Indica se o campo é virtual
```

The syntax for interface (View) is:

**AddField**( cIdField, cOrdem, cTitulo, cDescric, aHelp, cType, cPicture, bPictVar, cLookUp, ICanChange, cFolder, cGroup, aComboValues, nMaxLenCombo, cIniBrow, IVirtual, cPictVar, IInsertLine )

### Where:

<b>cIdField</b>	Field Name;
<b>cOrdem</b>	Order;
<b>cTitulo</b>	Field Title;
<b>cDescric</b>	Complete Description of the Field;
<b>aHelp</b>	Array with Help;
<b>cType</b>	Field Type;
<b>cPicture</b>	Picture;
<b>bPictVar</b>	PictureVar Block;
<b>cLookUp</b>	F3 Query;
<b>ICanChange</b>	Indicates whether the field is editable;

<b>cFolder</b>	Field folder;
<b>cGroup</b>	Field grouping;
<b>aComboValues</b>	List of field values allowed (Combo);
<b>nMaxLenCombo</b>	Maximum size of the largest combo option;
<b>cIniBrow</b>	Browse starter;
Indicates whether the field is virtual.	
<b>cPictVar</b>	Variable Picture;

We exemplified the use below:

```
Local oStruZA0 := FWFormStruct( 2, 'ZA0' )

oStruZA0:AddField( ;                               // Ord. Tipo Desc.
'ZA0_XEXEM1'                                     , ; // [01]  C   Nome do Campo
'50'                                             , ; // [02]  C   Ordem
AllTrim( 'Exemplo 1' )                          , ; // [03]  C   Título do campo
AllTrim( 'Campo Exemplo 1' )                    , ; // [04]  C   Descrição do campo
{ 'Exemplo de Campo de Manual 1' } , ; // [05]  A   Array com Help
'C'                                             , ; // [06]  C   Tipo do campo
'@!'                                           , ; // [07]  C   Picture
NIL                                             , ; // [08]  B   Bloco de Picture Var
''                                             , ; // [09]  C   Consulta F3
.T.                                             , ; // [10]  L   Indica se o campo é evitável
NIL                                             , ; // [11]  C   Pasta do campo
NIL                                             , ; // [12]  C   Agrupamento do campo

{ '1=Sim', '2=Não' }                           , ; // [13]  A   Lista de valores permitido do campo
(Combo)

NIL                                             , ; // [14]  N   Tamanho Maximo da maior opção do
combo

NIL                                             , ; // [15]  C   Inicializador de Browse
.T.                                             , ; // [16]  L   Indica se o campo é virtual
NIL                                             ) // [17]  C   Picture Variável
```

**Note:** The logical type are displayed as a checkbox in the interface (View)

## 10.5 Formatting the code block for structure (FWBuildFeature)

Some properties of the structure fields ask a specific construction of block code. After attributing or manipulating these properties, they must be informed in the standard the MVC expects.

After treating these priorities to use in applications, you must use the **FWBuildFeature** method to construct it.

**Example:**

```
FwBuildFeature( STRUCT_FEATURE_VALID,"Pertence('12')"
```

Where the first parameter indicates the property to be constructed and the second is the content to be assigned. The second parameter must always be or return a character type data.

The properties that must be regarded with this function are:

<b>STRUCT_FEATURE_VALID</b>	For <b>validation</b>
<b>STRUCT_FEATURE_WHEN</b>	For the <b>editing mode</b>
<b>STRUCT_FEATURE_INIPAD</b>	For the <b>standard starter</b>
<b>STRUCT_FEATURE_PICTVAR</b>	For <b>PictureVar</b>

The names of properties mentioned above are actually **#DEFINE**. To use this **#DEFINE** you must add the following policy in the source:

```
#INCLUDE 'FWMVCDEF.CH'
```

Note: Always use the **FWBuildFeature**. For the construction of the codomain properties, errors may occur in the application such as not updating the memory variables for form components.

## 10.6 Virtual MEMO type fields (FWMemoVirtual)

Some MEMO type fields use tables to save values (SYP<sup>3</sup>). These fields must be informed in the structure for the MVC to perform the treatment correctly.

For this, we use the **FWMemoVirtual** function.

**Example:**

```
FWMemoVirtual( oStruZA1,{ { 'ZA0_CDSYP1' , 'ZA0_MMSYP1' } , { 'ZA0_CDSYP2' , 'ZA0_MMSYP2' } } )
```

For these MEMO fields, there must always be another field including the code which the MEMO field was stored with in the auxiliary table

In the example, **oStruZA1** is the structure including the MEMO fields and the second parameter includes a two-dimensional vector where each pair related the structure field containing the code of the MEMO field with the MEMO itself.

If the auxiliary table to be used is not a SYP, a third parameter must be included in the two-

---

<sup>3</sup> SYP – Microsiga Protheus table that stores the field data of the virtual MEMO type

dimensional vector as the auxiliary table alias.

```
FWMemoVirtual( oStruZA1, { { 'ZA0_CDSYP1' , 'ZA0_MMSYP1', 'ZZ1' } , { 'ZA0_CDSYP2' ,  
'ZA0_MMSYP2' , 'ZZ1' } } )
```

**Note:** The MEMO field and the field storing the code must be part of the structure.

## ***10.7 Manual creation of trigger (AddTrigger / FwStruTrigger)***

If we wish to add a trigger in an already existing structure, we use the **AddTrigger** method.

The syntax is:

```
AddTrigger( cIdField, cTargetIdField, bPre, bSetValue )
```

Where:

**cIdField**                      Origin field name (ID);

**cTargetIdField**              Destination field name (ID);

**bPre**                          Code block of validation of the trigger execution;

**bSetValue**                      Code block of validation of the trigger execution;

The code blocks of this method are pending a specific construction. After assigning or manipulating, these properties must be informed in the standard that MVC expects.

To make the trigger construction easier, the **FwStruTrigger** function was created. It returns an array with 4 elements already formatted to be used in **AddTrigger**.

The syntax is:

**FwStruTrigger**              ( cDom, cCDom, cRegra, lSeek, cAlias, nOrdem, cChave, cCondic )

### **Where:**

**cDom**                          Domain field;

**cCDom**                        Codomain field;

**cRegra**                        Completion rule;

**lSeek**                        Positioned before the execution of triggers or not;

**cAlias**                        Table alias to be positioned;

**nOrdem**                        Order of the table to be positioned;

**cChave**                        Search key of the table to be positioned;

**cCondic**                        Condition for execution of the trigger;

Exemplifying:



```

Local oStruZA2 := FWFormStruct( 2, 'ZA2' )
Local aAux      := {}

aAux := FwStruTrigger(
'ZA2_AUTOR'      , ;
'ZA2_NOME'       , ;
'ZA0->ZA0_NOME' .. , ;
.T....., ;
'ZA0'....., ;
1....., ;
'xFilial("ZA0")+M->ZA2_AUTOR')

oStruct:AddTrigger( ;
aAux[1] , ; // [01] identificador (ID) do campo de origem
aAux[2] , ; // [02] identificador (ID) do campo de destino
aAux[3] , ; // [03] Bloco de código de validação da execução do gatilho
aAux[4] )   // [04] Bloco de código de execução do gatilho

```

## 10.8 Removing the polders of a structure (*SetNoFolder*)

If we wish to remove the folders that are configured in a structure, for example, when we use the **FWFormStruct** function, we use the **SetNoFolder** method. As the following:

```

Local oStruZA0 := FWFormStruct( 2, 'ZA0' )

// Retira as pastas da estrutura
oStruZA0:SetNoFolder()

```

## 10. 9 Removing the groupings of a structure field (*SetNoGroups*)

If we wish to remove the field groupings that are configured in a structure, for example, when we use the **FWFormStruct** function, we use the **SetNoGroups** method. As the following:

```

Local oStruZA0 := FWFormStruct( 2, 'ZA0' )
// Retira os agrupamentos de campos da estrutura
oStruZA0:SetNoGroups()

```

## 11.Creation of fields of total or counters (*AddCalc*)

In *MVC* you can create automatically a new composed component of totalizing or counting fields, a calculation component.

The calculation component fields are based in grid component of the model. When you update the grid component automatically, the calculation component fields are updated.

**Addcalc** is the data model (Model) in charge of this

The syntax is:

**AddCalc** (cId, cOwner, cIdForm, cIdField, cIdCalc, cOperation, bCond, bInitValue, cTitle, bFormula, nTamanho, nDecimal)

**Where:**

**cId** Identifier of the calculations component;

**cOwner** Identifier of superior component (*owner*). It is not necessarily the grid component where the data is from. The superior is usually the main **AddField** of the data model (*Model*);

**cIdForm** Grid component code that includes the field to which the field calculated refers;

**cIdField** Name of the grid component field to which the calculated field refers;

**cIdCalc** Identifier (name) for the calculated field;

**cOperation** Identifier of the operation to be carried out.

**The operations can be:**

**SUM** It sums up the grid component field;

**COUNT** Counts the grid component field;

**AVG** Makes the average of the grid component field;

**FORMULA** Runs a formula for the grid component field;

**bCond** Condition for evaluation of the calculated field. Receives the model object as parameter; Returning .T. (true), it runs the calculated field operations;

Example: {|oModel| teste (oModel)};

**bInitValue** Code block for initial value of the calculated field. Receives the model object as parameter;

Example: {|oModel| teste (oModel)};

**cTitle** Title for the field calculated;

**bFormula** Formula to be used when the cOperation parameter is FORMULA type. Received as parameters: the model object, the current value of the formula field, the content of the grid component content, logical field indicating if it is a sum (.T. (true)) or subtraction (.F. (false));

The value returned will be attributed to the calculated field;

**Example:**

```
{ |oModel, nTotalAtual, xValor, lSomando| Calculo( oModel, nTotalAtual, xValor, lSomando ) };
```

**nTamanho** Size of the calculated field (if not informed, use standard size). The standard sizes for operations are:

**SUM** is the size of a grid component field + 3;

If the grid component field has a size 9, the calculated field has 12.

**COUNT** is the size fixed in 6;

**AVG** is the size of the grid component field. If the grid component field has a size 9, the calculated field has 9;

**FORMULA** is the size of the grid component field + 3. If the grid component field has a size 9, the calculated field has 12;

**nDecimal** Number of decimal places of the calculated field;

**Note:** For the **SUM** and **AVG** operations, the grid component field has to be numeric.

### Example:

```
Static Function ModelDef()
...
oModel:AddCalc( 'COMP022CALC1', 'ZA1MASTER', 'ZA2DETAIL', 'ZA2_AUTOR', 'ZA2__TOT01',
'COUNT', { | oFW | COMP022CAL( oFW, .T. ) },,'Total Pares' )
oModel:AddCalc( 'COMP022CALC1', 'ZA1MASTER', 'ZA2DETAIL', 'ZA2_AUTOR', 'ZA2__TOT02',
'COUNT', { | oFW | COMP022CAL( oFW, .F. ) },,'Total Impares' )
...
```

### Where:

**COMP022CALC1** is the calculation component identifier;

**ZA1MASTER** is the superior component (owner) identifier;

**ZA2DETAIL** is the grid component code from which data will come;

**ZA2\_AUTOR** is the name of the grid component field to which the calculated field refers;

**ZA2\_\_TOT01** is the identifier (name) for the calculated field;

**COUNT** is the Identifier of the operation to be carried out;

**{ | oFW | COMP022CAL( oFW, .T. ) }** is the condition for evaluation of the calculated field;

**'Total Pairs'** is the title for the calculated fields;

In **ViewDef**, we also have to define the calculation component. The data used in a calculation component are based in a grid component but its display is the same as a form component as we use **AddField** for the calculation component. To obtain the structure created in **ModelDef**, we use **FWCalcStruct**.

**Example:**

```
Static Function View
...
// Cria o objeto de Estrutura
oCalc1 := FWCalcStruct( oModel:GetModel( 'COMP022CALC1' ) )
//Adiciona no nosso View um controle do tipo FormGrid(antiga newgetdados)
oView:AddField( 'VIEW_CALC', oCalc1, 'COMP022CALC1' )
...
```

## 12. Other functions for *MVC*

Some functions can be especially useful during the development of an application.

### 12.1 Interface direct execution (*FWExecView*)

It executed the interface (View) with an established operation.

This function instantiates the interface (View) and therefore the data model (Model) with the **view, add, change or delete** operations. Its intention is to perform similarly to the **AXVISUAL, AXINCLI, AXALTERA** and **AXDELETA** functions.

The syntax is:

**FWExecView** (cTitulo, cPrograma, nOperation, oDlg, bCloseOnOk, bOk, nPercReducao, aEnableButtons, bCancel )

#### Where:

<b>cTitulo</b>	Window title;
<b>cPrograma</b>	Name of source-program;
<b>nOperation</b>	Indicates the operation code (inclusion, editing or deletion);
<b>oDlg</b>	Window object in which the View must be put. If not informed, a new window is created;
<b>bCloseOnOK</b>	indicates if the window must be closed at the end of the operation. If it returns .T. (true) close the window;
<b>bOk</b>	Block executed when enabling the confirm button returning .F. (false) hinders the window closing;
<b>nPercReducao</b>	If informed, it reduces the window percentually;
<b>aEnableButtons</b>	Indicates the buttons of the button bar that are enabled;
<b>bCancel</b>	Block executed when enabling the cancel button which hinders the window closing if it returns .F. (false);

#### **The return of this function is:**

- 0 If the user concludes the operation with the confirm button;
- 1 If the user concludes the operation with the cancel button;

**Example:**

```

lok := ( FWExecView('Inclusão por FWExecView','COMP011_MVC', MODEL_OPERATION_INSERT,, {
      || .T. } ) == 0 )

If lok
    Help( ,, 'Help',, 'Foi confirmada a operação, 1, 0 )
Else
    Help( ,, 'Help',, 'Foi cancelada a operação, 1, 0 )
EndIf

```

**12.2 Active data model (FWModelActive)**

In an application, we can work with more than one data model (*Model*). The **FWModelActive** function returns the data model (*Model*) which is active at the moment.

**Example:**

```

Static Function COMP021BUT()
Local oModel      := FWModelActive()
Local nOperation := oModel:GetOperation()

```

To define the active data model (Model):

```

Local oModelBkp := FWModelActive()
...
FWModelActive( oModelBkp )

```

**12.3 Active interface (FWViewActive)**

In an application we can work with more than one interface (View). The **FWViewActive** function returns the interface (View) active at the time.

**Example:**

```

Static Function COMP021BUT()
Local oView      := FWViewActive()
oView:Refresh()

```

To define the interface (View):

```

Local oViewBkp := FWViewActive()
...
FWViewActive(oViewBkp)

```

**12.4 Load the data model of an existing application (FWLoadModel)**

To create an object with a data model of an application, we use the **FWLoadModel** function.

The syntax is:

**FWLoadModel**( <source name> )

**Example:**

```
Static Function ModelDef()  
// Utilizando um model que ja existe em outra aplicacao  
Return FWLoadModel( 'COMP011_MVC' )
```

## 12.5 Load the interface of an existing application (FWLoadView)

To create an object with the data model of an application, we use the **FWLoadView** function.

The syntax is:

**FWLoadView** ( <source name> )

**Example:**

```
Static Function ViewDef()  
// Utilizando uma view que ja existe em outra aplicacao  
Return FWLoadView( 'COMP011_MVC' )
```

## 12.6 Load a menu of an existing application (FWLoadMenudef)

To create a vector with the menu options of an application, we use **FWLoadMenudef** function.

The syntax is:

**FWLoadMenudef** ( <nome do fonte> )

**Example:**

```
Static Function MenuDef()  
// Utilizando um menu que ja existe em outra aplicacao  
Return FWLoadMenuDef( 'COMP011_MVC' )
```

## 12.7 Obtaining standard menu (FWMVCMenu)

We can create a menu with the standard options for MVC using the **FWMVCMENU** function.

The syntax is:

**FWMVCMENU** ( <source name> )

**Example:**

```
//-----  
  
Static Function MenuDef()
```

```
Return FWMVCMenu( 'COMP011_MVC' ) )
```

A standard menu is created with the following options: **View, Add, Edit, Delete, Print and Copy.**

## 13. Browse with a markup column (FWMarkBrowse)

If we wish to construct an application with a *Browse* that uses a markup column, similar to the **MarkBrowse** function in traditional AdvPL, we use the **FWMarkBrowse** class.

Just as **FWmBrowse** (see chapter 0 3. Applications with Browsers (FWMBrowse)), **FWMarkBrowse** is not exclusively of MVC and can be used by applications that does not use it.

In this content we will not explore the **FWMarkBrowse** resources, we will learn about its main functions and features to use in applications with MVC.

As premise, there must be a character-type field in the table with size 2, which receives the mark physically. A different mark is generated each time **FWMarkBrowse** is executed.

We will begin the basic construction of **FWMarkBrowse**.

First, you must create a **Browse** as the following:

```
oMark := FWMarkBrowse():New()
```

We defined the table to be displayed on the **Browse** by the **SetAlias** method. The columns, orders, etc are obtained through the metadata (dictionaries) for display

```
oMark:SetAlias('ZA0')
```

We defined the title to be displayed as **SetDescription** method.

```
oMark:SetDescription('Seleção do Cadastro de Autor/Interprete')
```

We define the table field that received the physical mark.

```
oMark:SetFieldMark( 'ZA0_OK' )
```

And in the end activate the class.

```
oMark:Activate()
```

With this basic structure, we built an application with Browse.

But for now, we only have the **Browse** with markup column, we must define an action for the marked items. For this, we can add the function treating the marked items in **MenuDef**.

```
ADD OPTION aRotina TITLE 'Processar' ACTION 'U_COMP25PROC()' OPERATION 2 ACCESS 0
```

In the function treating the marked items, you must identify if a record is marked or not. To know the mark being used at the time, we use the **Mark** method.

```
Local cMarca := oMark:Mark()
```



To know if the record is marked we use the **IsMark** method passing the mark as parameter.

```
If oMark:IsMark(cMarca)
```

You can also add other options such as View or Edit in the option menu (**MenuDef**), but you must also create the data model (Model) and the interface (View).

All other **FWMBrowse** features also apply to **FWMarkBrowse** such as captions, filters, details, etc.

A resource that **FWMarkBrowse** has is the exclusive markup control of a record by the user.

If 2 users open the same **FWMarkBrowse** and try to mark the same record, **FWMarkBrowse** allows that only one of them carries out the markup. To enable this feature, we use the **SetSemaphore** method .

Below is an example of **FWMarkBrowse** use

```
User Function COMP025_MVC()
Private oMark

// Instanciamento do classe
oMark := FWMarkBrowse():New()

// Definição da tabela a ser utilizada
oMark:SetAlias('ZA0')

// Define se utiliza controle de marcação exclusiva do oMark:SetSemaphore(.T.)

// Define a titulo do browse de marcacao
oMark:SetDescription('Seleção do Cadastro de Autor/Interprete')

// Define o campo que sera utilizado para a marcação
oMark:SetFieldMark( 'ZA0_OK' )

// Define a legenda
oMark:AddLegend( "ZA0_TIPO=='1'", "YELLOW", "Autor"      )
oMark:AddLegend( "ZA0_TIPO=='2'", "BLUE"   , "Interprete"  )

// Definição do filtro de aplicacao
oMark:SetFilterDefault( "ZA0_TIPO=='1'" )

// Ativacao da classe
oMark:Activate()
```

```

Return NIL

//-----
Static Function MenuDef()
Local aRotina := {}

ADD OPTION aRotina TITLE 'Visualizar' ACTION 'VIEWDEF.COMP025_MVC' OPERATION 2 ACCESS 0
ADD OPTION aRotina TITLE 'Processar' ACTION 'U_COMP25PROC()' OPERATION 2 ACCESS 0

Return aRotina

//-----
Static Function ModelDef()
// Utilizando um model que ja existe em outra aplicacao
Return FWLoadModel( 'COMP011_MVC' )

//-----
Static Function ViewDef()
// Utilizando uma View que ja existe em outra aplicacao
Return FWLoadView( 'COMP011_MVC' )

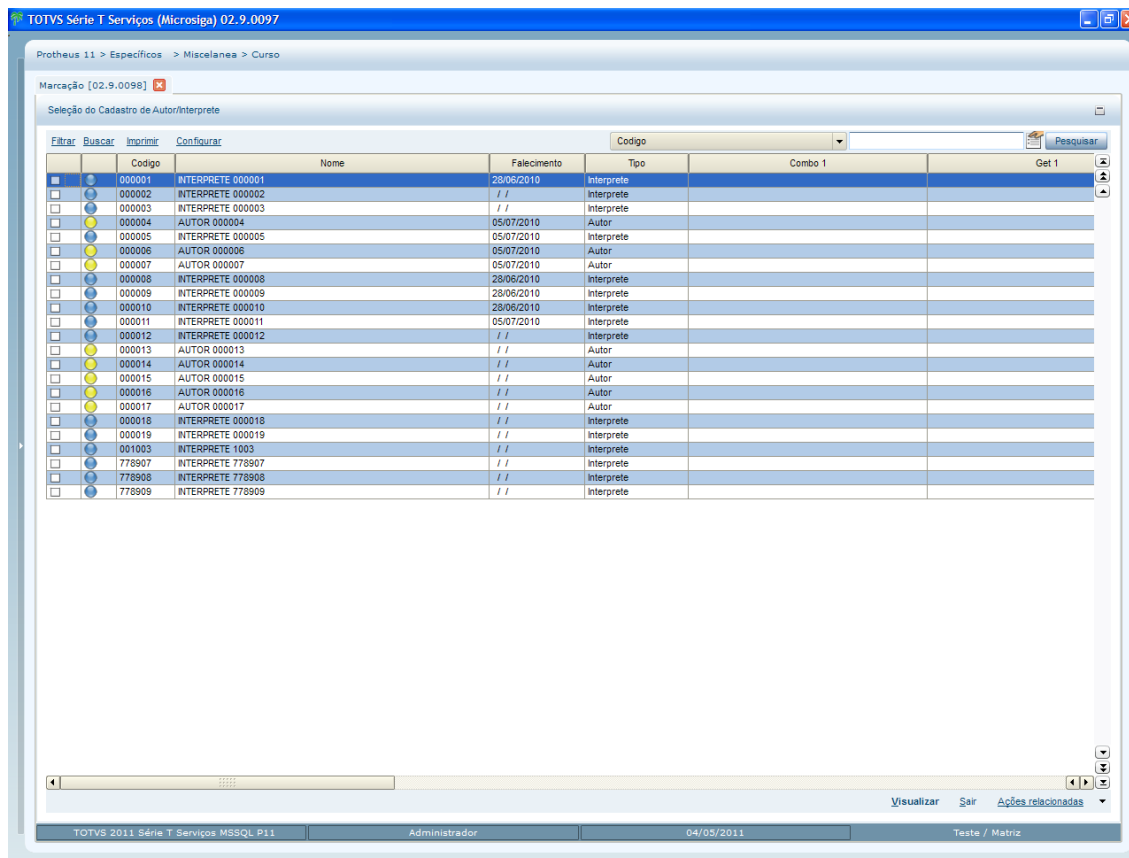
//-----
User Function COMP25PROC()
Local aArea := GetArea()
Local cMarca := oMark:Mark()
Local nCt := 0
ZA0->( dbGoTop() )
While !ZA0->( EOF() )
    If oMark:IsMark(cMarca)
        nCt++
    EndIf
    ZA0->( dbSkip() )
End

ApmMsgInfo( 'Foram marcados ' + AllTrim( Str( nCt ) ) + ' registros.' )
RestArea( aArea )

Return NIL

```

Visually we have:



## 14. Multiple Browsers

By using the **FWmBrowse** class we can write applications with more than one object of this class, that is, we can write applications to work with multiple **Browsers**.

For example, we can develop an application for sales order in which we have a **Browse** with the header of items and another with the items in the same screen and as we navigate through the header **Browse** records, the items are automatically updated in the other **Browse**.

For this, create 2 **FWmBrowse** objects in our application and relate them among themselves. Below we describe how to do this. We create an application with 3 **Browsers**.

First, we create a common **Dialog** screen, each one of the **Browsers** must be **anchored** in a **container** object. For this, we use **FWLayer** with 2 lines and in one of these lines we add 2 columns.

For more details on **FWLayer**, check the specific documentation on TDN<sup>4</sup>.

```
User Function COMP024_MVC()

Local aCoors := FWGetDialogSize( oMainWnd )

Local oPanelUp, oFWLayer, oPanelLeft, oPanelRight, oBrowseUp, oBrowseLeft, oBrowseRight,
oRelacZA4, oRelacZA5

Define MsDialog oDlgPrinc Title 'Multiplos FWmBrowse' From aCoors[1], aCoors[2] To aCoors[3],
aCoors[4] Pixel

//

// Cria o container onde serão colocados os browsers
//

oFWLayer := FWLayer():New()
oFWLayer:Init( oDlgPrinc, .F., .T. )

//

// Define PaineL Superior
//

oFWLayer:AddLine( 'UP', 50, .F. ) // Cria uma "linha" com 50% da tela
oFWLayer:AddCollumn( 'ALL', 100, .T., 'UP' ) // Na "linha" criada eu crio uma
coluna com 100% da tamanho dela
oPanelUp := oFWLayer:GetColPanel( 'ALL', 'UP' ) // Pego o objeto desse pedaço do
container

//
```

---

<sup>4</sup> TDN – TOTVS Developer Network portal focused on Microsiga Protheus developers

```
// PaineL Inferior
//
oFWLayer:AddLine( 'DOWN', 50, .F. )           // Cria uma "linha" com 50% da tela
oFWLayer:AddCollumn( 'LEFT' , 50, .T., 'DOWN' ) // Na "linha" criada eu crio uma
coluna com 50% da tamanho dela
oFWLayer:AddCollumn( 'RIGHT', 50, .T., 'DOWN' ) // Na "linha" criada eu crio uma
coluna com 50% da tamanho dela
oPanelLeft := oFWLayer:GetColPanel( 'LEFT' , 'DOWN' ) // Pego o objeto do pedaço esquerdo
oPanelRight := oFWLayer:GetColPanel( 'RIGHT', 'DOWN' ) // Pego o objeto do pedaço direito
```

After this, we create 3 **Browses** as described in the chapter **0 3. Applications with Browsets (FWMBrowse)**.

This is the **1st Browse**.

```
*/
// FwMBrowse Superior Albuns
//
oBrowseUp:= FwMBrowse():New()
oBrowseUp:SetOwner( oPanelUp )           // Aqui se associa o browse ao
                                           //componente de tela
oBrowseUp:SetDescription( "Albuns" )
oBrowseUp:SetAlias( 'ZA3' )
oBrowseUp:SetMenuDef( 'COMP024_MVC' )    // Define de onde virao os
                                           // botoes deste browse
oBrowseUp:SetProfileID( '1' )             // identificador (ID) para o Browse
oBrowseUp:ForceQuitButton()              // Força exibição do botão
                                           // Sair
oBrowseUp:Activate()
```

Notice 2 methods defined in this **Browse: SetProfileID** and **ForceQuitButton**

The **SetProfileID** method defines an identifier (ID) for **Browse**, this is necessary since we already have more than one **Browse** in the application.

The **ForceQuitButton** method displays the **Exit** button in the following **Browse** options. As there is more than one **Browse**, the **Exit** button us automatically in none of them. This method makes it appear in the **Browse**.

Notice also that we use the **SetMenuDef** method to define what source must be used to obtain **MenuDef**. When we do not use the **SetMenuDef** method automatically, the **Browse** searches in the source itself where it finds the **Menudef** to be used.

These are **the 2nd and 3rd Browsers**:

```
oBrowseLeft:= FWMBrowse():New()
oBrowseLeft:SetOwner( oPanelLeft )
oBrowseLeft:SetDescription( 'Musicas' )
oBrowseLeft:SetMenuDef( '' ) // Referencia vazia para que nao
                               // exiba nenhum botão
oBrowseLeft:DisableDetails()
oBrowseLeft:SetAlias( 'ZA4' )
oBrowseLeft:SetProfileID( '2' )
oBrowseLeft:Activate()

oBrowseRight:= FWMBrowse():New()
oBrowseRight:SetOwner( oPanelRight )
oBrowseRight:SetDescription( 'Autores/Interpretes' )
oBrowseRight:SetMenuDef( '' ) // Referencia vazia para que nao funcao
                               // exiba nenhum botao
oBrowseRight:DisableDetails()
oBrowseRight:SetAlias( 'ZA5' )
oBrowseRight:SetProfileID( '3' )
oBrowseRight:Activate()
```

Notice that in these **Browsers** we use the **SetMenuDef** method with an empty reference. Since we wish only the main **Browse** has the action buttons, if we do not **SetMenuDef** automatically, the **Browse** searches the source where it is and with an empty reference, buttons are not displayed.

Now that we defined the **Browsers**, we must relate them so that when we transfer one, the others are updated automatically.

To create the relationship, we use the **FWBrwRelation** class. Similarly to the relationships among entities made in the data model (Model), you must enter the relationship keys from the **parent** to the **parent**.

We instantiate **FWBrwRelation** and we use the **AddRelation** method.

The syntax of this **FWBrwRelation** method is:

**AddRelation**( <ParentBrowse>, <SecondaryBrowse>, <Vector with relationship fields> )

As we have 3 Browsers, we have 2 relationships:

```
oRelacZA4:= FWBrwRelation():New()
oRelacZA4:AddRelation( oBrowseUp , oBrowseLeft , { { 'ZA4_FILIAL', 'xFilial( "ZA4" )' }, {
'ZA4_ALBUM' , 'ZA3_ALBUM' } } )
oRelacZA4:Activate()
```

```
oRelacZA5:= FWBrwRelation():New()
oRelacZA5:AddRelation( oBrowseLeft, oBrowseRight, { { 'ZA5_FILIAL', 'xFilial( "ZA5" )' }, {
'ZA5_ALBUM' , 'ZA4_ALBUM' }, { 'ZA5_MUSICA', 'ZA4_MUSICA' } } )
oRelacZA5:Activate()
```

Here is a complete example of application with **Browses** method:

```
User Function COMP024_MVC()
Local aCoors := FWGetDialogSize( oMainWnd )
Local oPanelUp, oFWLayer, oPanelLeft, oPanelRight, oBrowseUp, oBrowseLeft, oBrowseRight,
oRelacZA4, oRelacZA5

Private oDlgPrinc

Define MsDialog oDlgPrinc Title 'Multiplos FwMBrowse' From aCoors[1], aCoors[2] To aCoors[3],
aCoors[4] Pixel

//
// Cria o container onde serão colocados os browses

//
oFWLayer := FWLayer():New()
oFWLayer:Init( oDlgPrinc, .F., .T. )

//
// Define PaineL Superior
//
oFWLayer:AddLine( 'UP', 50, .F. )
// Cria uma "linha" com 50% da tela
oFWLayer:AddCollumn( 'ALL', 100, .T., 'UP' )
// Na "linha" criada eu crio uma coluna com 100% da tamanho dela
oPanelUp := oFWLayer:GetColPanel( 'ALL', 'UP' )
// Pego o objeto desse pedaço do container

//
// PaineL Inferior
//
oFWLayer:AddLine( 'DOWN', 50, .F. )
// Cria uma "linha" com 50% da tela
oFWLayer:AddCollumn( 'LEFT' , 50, .T., 'DOWN' )
// Na "linha" criada eu crio uma coluna com 50% da tamanho dela
oFWLayer:AddCollumn( 'RIGHT', 50, .T., 'DOWN' )
// Na "linha" criada eu crio uma coluna com 50% da tamanho dela
```

```

oPanelLeft := oFWLayer:GetColPanel( 'LEFT' , 'DOWN' ) // Pegou o objeto do pedaço esquerdo
oPanelRight := oFWLayer:GetColPanel( 'RIGHT', 'DOWN' ) // Pegou o objeto do pedaço direito

//
// FwMBrowse Superior Albuns
//
oBrowseUp:= FwMBrowse():New()
oBrowseUp:SetOwner( oPanelUp )
// Aqui se associa o browse ao componente de tela
oBrowseUp:SetDescription( "Albuns" )
oBrowseUp:SetAlias( 'ZA3' )
oBrowseUp:SetMenuDef( 'COMP024_MVC' )
// Define de onde virao os botoes deste browse
oBrowseUp:SetProfileID( '1' )
oBrowseUp:ForceQuitButton()
oBrowseUp:Activate()

//
// Lado Esquerdo Musicas
//
oBrowseLeft:= FwMBrowse():New()
oBrowseLeft:SetOwner( oPanelLeft )
oBrowseLeft:SetDescription( 'Musicas' )
oBrowseLeft:SetMenuDef( '' )
// Referencia vazia para que nao exiba nenhum botao
oBrowseLeft:DisableDetails()
oBrowseLeft:SetAlias( 'ZA4' )
oBrowseLeft:SetProfileID( '2' )
oBrowseLeft:Activate()

//
// Lado Direito Autores/Interpretes
//
oBrowseRight:= FwMBrowse():New()
oBrowseRight:SetOwner( oPanelRight )
oBrowseRight:SetDescription( 'Autores/Interpretes' )
oBrowseRight:SetMenuDef( '' )
// Referencia vazia para que nao exiba nenhum botao
oBrowseRight:DisableDetails()
oBrowseRight:SetAlias( 'ZA5' )

```



```

oBrowseRight.SetProfileID( '3' )
oBrowseRight.Activate()

//
// Relacionamento entre os Paineis
/

oRelacZA4:= FWBrwRelation():New()
oRelacZA4.AddRelation( oBrowseUp , oBrowseLeft , { { 'ZA4_FILIAL', 'xFilial( "ZA4" )' }, {
'ZA4_ALBUM' , 'ZA3_ALBUM' } } )
oRelacZA4.Activate()

oRelacZA5:= FWBrwRelation():New()
oRelacZA5.AddRelation( oBrowseLeft, oBrowseRight, { { 'ZA5_FILIAL', 'xFilial( "ZA5" )' }, {
'ZA5_ALBUM' , 'ZA4_ALBUM' }, { 'ZA5_MUSICA', 'ZA4_MUSICA' } } )
oRelacZA5.Activate()

Activate MsDialog oDlgPrinc Center

Return NIL

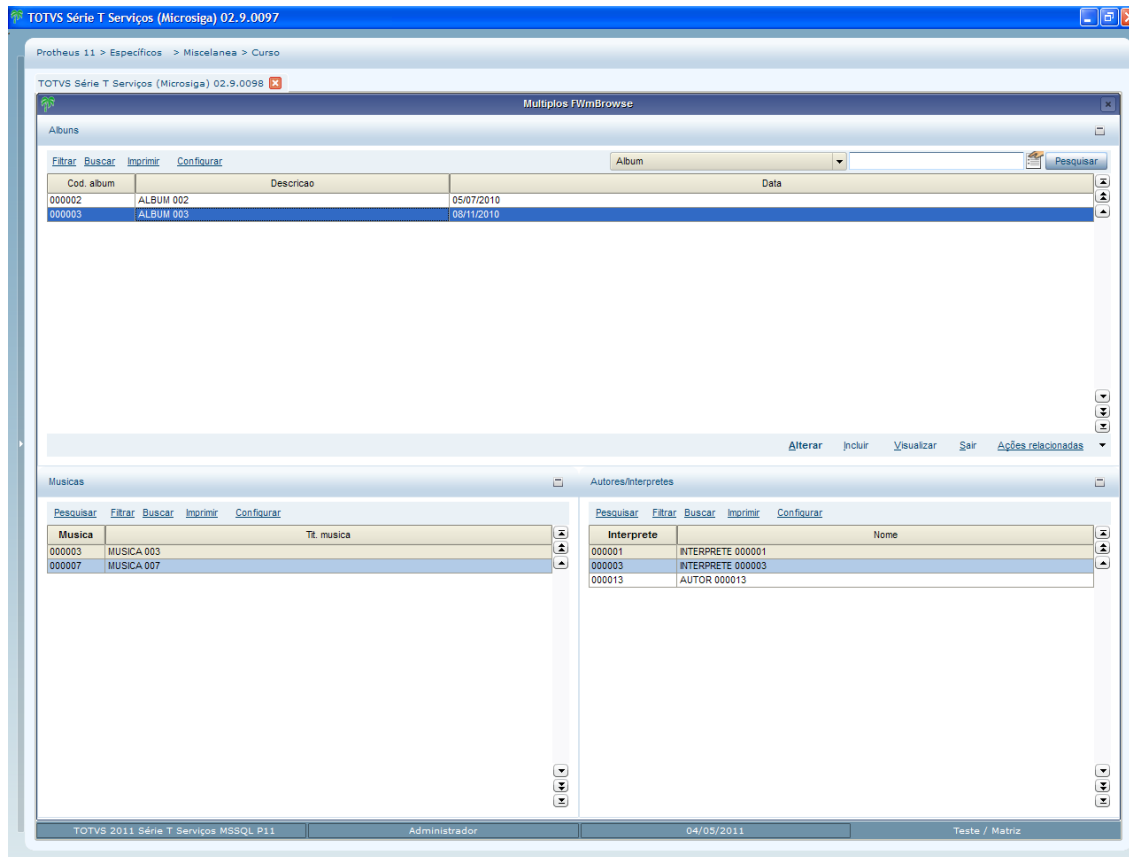
//-----
Static Function MenuDef()
Return FWLoadMenuDef( 'COMP023_MVC' )

//-----
Static Function ModelDef()
// Utilizamos um model que ja existe
Return FWLoadModel( 'COMP023_MVC' )

//-----
Static Function ViewDef()
// Utilizamos uma View que ja existe
Return FWLoadView( 'COMP023_MVC' )

```

Visually we have:



## 15. Automatic routine

When an application is developed by the use of the MVC concept and classes, you can use the data model in other applications, similar to what an **automatic routine** would be in the traditional development.

The use of the **MSExecAuto** function does not exist anymore; The basic idea is to instantiate the data model (*Model*) you want, assign the values and validate them.

For better understanding, we use the source below as an example, in which what would be an **automatic routine** for import of a simple record.

Notice the comments.

```
//-----
// Rotina principal de Importação
//-----

User Function COMP031_MVC()

Local   aSay      := {}
Local   aButton   := {}
Local   nOpc      := 0
Local   Titulo    := 'IMPORTACAO DE COMPOSITORES'
Local   cDesc1    := 'Esta rotina fará a importação de compositores/interpretes'
Local   cDesc2    := 'conforme layout.'
Local   cDesc3    := ''
Local   lOk       := .T.

aAdd( aSay, cDesc1 )
aAdd( aSay, cDesc2 )
aAdd( aSay, cDesc3 )

aAdd( aButton, { 1, .T., { || nOpc := 1, FechaBatch() } } )
aAdd( aButton, { 2, .T., { || FechaBatch() } } )
FormBatch( Titulo, aSay, aButton )

If nOpc == 1
    Processa( { || lOk := Runproc() }, 'Aguarde', 'Processando...', .F.)

    If lOk
        ApmMsgInfo( 'Processamento terminado com sucesso.', 'ATENÇÃO' )
    Else
        ApmMsgStop( 'Processamento realizado com problemas.', 'ATENÇÃO' )
    EndIf
EndIf
```

```

EndIf

Return NIL

//-----
// Rotina Auxiliar de Importação
//-----

Static Function Runproc()
Local lRet      := .T.
Local aCampos := {}

// Criamos um vetor com os dados para facilitar o manuseio dos dados
aCampos := {}
aAdd( aCampos, { 'ZA0_CODIGO', '000100'      } )
aAdd( aCampos, { 'ZA0_NOME'   , 'Vila Lobos'   } )
aAdd( aCampos, { 'ZA0_NOTAS'  , 'Observações...' } )
aAdd( aCampos, { 'ZA0_TIPO'   , 'C'           } )

If !Import( 'ZA0', aCampos )
    lRet := .F.
EndIf

// Importamos outro registro
aCampos := {}
aAdd( aCampos, { 'ZA0_CODIGO', '000102'      } )
aAdd( aCampos, { 'ZA0_NOME'   , 'Tom Jobim'   } )
aAdd( aCampos, { 'ZA0_NOTAS'  , 'Observações...' } )
aAdd( aCampos, { 'ZA0_TIPO'   , 'C'           } )

If !Import( 'ZA0', aCampos )
    lRet := .F.
EndIf

// Importamos outro registro
aCampos := {}
aAdd( aCampos, { 'ZA0_CODIGO', '000104'      } )
aAdd( aCampos, { 'ZA0_NOME'   , 'Emilio Santiago' } )
aAdd( aCampos, { 'ZA0_NOTAS'  , 'Observações...' } )
aAdd( aCampos, { 'ZA0_TIPO'   , 'I'           } )

If !Import( 'ZA0', aCampos )

```

```

        lRet := .F.
    EndIf
    Return lRet

//-----
// Importação dos dados
//-----

Static Function Import( cAlias, aCampos )
    Local oModel, oAux, oStruct
    Local nI      := 0
    Local nPos     := 0
    Local lRet     := .T.
    Local aAux     := {}
    dbSelectArea( cAlias )
    dbSetOrder( 1 )

    // Aqui ocorre o instanciamento do modelo de dados (Model)
    // Neste exemplo instanciamos o modelo de dados do fonte COMP011_MVC
    // que é a rotina de manutenção de compositores/interpretes
    oModel := FWLoadModel( 'COMP011_MVC' )

    // Temos que definir qual a operação deseja: 3 - Inclusão / 4 - Alteração / 5 - Exclusão
    oModel:SetOperation( 3 )

    // Antes de atribuírmos os valores dos campos temos que ativar o modelo
    oModel:Activate()

    // Instanciamos apenas referentes às dados
    oAux := oModel:GetModel( cAlias + 'MASTER' )

    // Obtemos a estrutura de dados
    oStruct := oAux:GetStruct()
    aAux := oStruct:GetFields()

    For nI := 1 To Len( aCampos )
        // Verifica se os campos passados existem na estrutura do modelo
        If ( nPos := aScan(aAux, {|x| AllTrim( x[3] )== AllTrim(aCampos[nI][1]) } ) ) > 0
            // É feita a atribuição do dado ao campo do Model
            If !( lAux := oModel:SetValue( cAlias + 'MASTER', aCampos[nI][1], aCampos[nI][2] ) )
        ) )
    
```

```

// Caso a atribuição não possa ser feita, por algum motivo (validação, por
exemplo)

// o método SetValue retorna .F.
lRet      := .F.
Exit

EndIf

EndIf

Next nI

If lRet
// Faz-se a validação dos dados, note que diferentemente das tradicionais
// "rotinas automáticas"
// neste momento os dados não são gravados, são somente validados.
If ( lRet := oModel:VldData() )
// Se o dados foram validados faz-se a gravação efetiva dos dados (commit)
oModel:CommitData()
EndIf
EndIf
If !lRet
// Se os dados não foram validados obtemos a descrição do erro para gerar LOG ou
mensagem de aviso
aErro      := oModel:GetErrorMessage()
// A estrutura do vetor com erro é:
// [1] identificador (ID) do formulário de origem
// [2] identificador (ID) do campo de origem
// [3] identificador (ID) do formulário de erro
// [4] identificador (ID) do campo de erro
// [5] identificador (ID) do erro
// [6] mensagem do erro
// [7] mensagem da solução
// [8] Valor atribuído
// [9] Valor anterior

AutoGrLog( "Id do formulário de origem:" + ' [' + AllToChar( aErro[1] ) + ']' )
AutoGrLog( "Id do campo de origem:      " + ' [' + AllToChar( aErro[2] ) + ']' )
AutoGrLog( "Id do formulário de erro:   " + ' [' + AllToChar( aErro[3] ) + ']' )
AutoGrLog( "Id do campo de erro:        " + ' [' + AllToChar( aErro[4] ) + ']' )
AutoGrLog( "Id do erro:                  " + ' [' + AllToChar( aErro[5] ) + ']' )
AutoGrLog( "Mensagem do erro:           " + ' [' + AllToChar( aErro[6] ) + ']' )
AutoGrLog( "Mensagem da solução:        " + ' [' + AllToChar( aErro[7] ) + ']' )

```

```

        AutoGrLog( "Valor atribuído:          " + ' [' + AllToChar( aErro[8]  ) + ']' )
        AutoGrLog( "Valor anterior:          " + ' [' + AllToChar( aErro[9]  ) + ']' )
        MostraErro()
    EndIf

// Desativamos o Model
oModel:DeActivate()

Return lRet

```

In this other example, we have the import to a data model where there is a **Master-Detail** structure (**Parent-Secondary**). Also, what we do is to instantiate the data model (Model) that we want, assign the values to it and validate it, but we do this for two entities.

Notice the comments:

```

//-----
// Rotina principal de Importação
//-----
User Function COMP032_MVC()
Local  aSay      := {}
Local  aButton   := {}
Local  nOpc      := 0
Local  Titulo    := 'IMPORTACAO DE MUSICAS'
Local  cDesc1    := 'Esta rotina fará a importação de musicas'
Local  cDesc2    := 'conforme layout.'
Local  cDesc3    := ''
Local  lOk       := .T.

aAdd( aSay, cDesc1 )
aAdd( aSay, cDesc2 )
aAdd( aSay, cDesc3 )

aAdd( aButton, { 1, .T., { || nOpc := 1, FechaBatch() } } )
aAdd( aButton, { 2, .T., { || FechaBatch() } } )

FormBatch( Titulo, aSay, aButton )
If nOpc == 1
    Processa( { || lOk := Runproc() }, 'Aguarde', 'Processando...', .F.)
        If lOk
            ApmMsgInfo( 'Processamento terminado com sucesso.', 'ATENÇÃO' )
        Else

```

```

        ApMsgStop( 'Processamento realizado com problemas.', 'ATENÇÃO' )
    EndIf
EndIf
Return NIL

//-----
// Rotina auxiliar de Importação
//-----

Static Function Runproc()
Local lRet      := .T.
Local aCposCab := {}
Local aCposDet := {}
Local aAux      := {}

// Criamos um vetor com os dados de cabeçalho e outro para itens para facilitar o
manuseio dos dados
aCposCab := {}
aCposDet := {}
aAdd( aCposCab, { 'ZA1_TITULO' , 'LA, LA, LA,' } )
aAdd( aCposCab, { 'ZA1_DATA', Date() } )

aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '01' } )
aAdd( aAux, { 'ZA2_AUTOR', '000100' } )
aAdd( aCposDet, aAux )

aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '02' } )
aAdd( aAux, { 'ZA2_AUTOR', '000104' } )
aAdd( aCposDet, aAux )
If !Import( 'ZA1', 'ZA2', aCposCab, aCposDet )
    lRet := .F.
EndIf

// Importamos outro conjunto de dados
aCposCab := {}
aCposDet := {}
aAdd( aCposCab, { 'ZA1_TITULO' , 'BLA, BLA, BLA' } )
aAdd( aCposCab, { 'ZA1_DATA', Date() } )

```



```

aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '01' } )
aAdd( aAux, { 'ZA2_AUTOR', '000102' } )
aAdd( aCposDet, aAux )
aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '02' } )
aAdd( aAux, { 'ZA2_AUTOR', '000104' } )
aAdd( aCposDet, aAux )

If !Import( 'ZA1', 'ZA2', aCposCab, aCposDet )
    lRet := .F.
EndIf

// Importamos outro conjunto de dados
aCposCab := {}
aCposDet := {}
aAdd( aCposCab, { 'ZA1_TITULO' , 'ZAP, ZAP, ZAP' } )
aAdd( aCposCab, { 'ZA1_DATA', Date() } )

aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '01' } )
aAdd( aAux, { 'ZA2_AUTOR', '000100' } )
aAdd( aCposDet, aAux )

aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '02' } )
aAdd( aAux, { 'ZA2_AUTOR', '000102' } )
aAdd( aCposDet, aAux )

If !Import( 'ZA1', 'ZA2', aCposCab, aCposDet )
    lRet := .F.
EndIf

Return lRet

//-----
// Importacao dos dados
//-----
Static Function Import( cMaster, cDetail, aCpoMaster, aCpoDetail )
Local oModel, oAux, oStruct

```

```

Local  nI          := 0
Local  nJ          := 0
Local  nPos        := 0
Local  lRet        := .T.
Local  aAux        := {}
Local  aC          := {}
Local  aH          := {}
Local  nItErro     := 0
Local  lAux        := .T.

dbSelectArea( cDetail )
dbSetOrder( 1 )

dbSelectArea( cMaster )
dbSetOrder( 1 )

// Aqui ocorre o instanciamento do modelo de dados (Model)
// Neste exemplo instanciamos o modelo de dados do fonte COMP022_MVC
// que é a rotina de manutenção de musicas
oModel := FWLoadModel( 'COMP022_MVC' )

// Temos que definir qual a operação deseja: 3 - Inclusão / 4 - Alteração / 5 - Exclusão
oModel:SetOperation( 3 )

// Antes de atribuírmos os valores dos campos temos que ativar o modelo
oModel:Activate()

// Instanciamos apenas a parte do modelo referente aos dados de cabeçalho
oAux    := oModel:GetModel( cMaster + 'MASTER' )

// Obtemos a estrutura de dados do cabeçalho
oStruct := oAux:GetStruct()
aAux    := oStruct:GetFields()

If lRet

    For nI := 1 To Len( aCpoMaster )

// Verifica se os campos passados existem na estrutura do cabeçalho

        If ( nPos := aScan( aAux, { |x| AllTrim( x[3] ) == AllTrim(
aCpoMaster[nI][1] ) } ) ) > 0

```

```

        // É feita a atribuição do dado aos campo do Model do cabeçalho
        If !( lAux := oModel:SetValue( cMaster + 'MASTER', aCpoMaster[nI][1],
aCpoMaster[nI][2] ) )

        // Caso a atribuição não possa ser feita, por algum motivo (validação,
por exemplo)

        // o método SetValue retorna .F.
        lRet      := .F.
        Exit
    EndIf
EndIf

Next

EndIf
If lRet
    // Instanciamos apenas a parte do modelo referente aos dados do item
    oAux      := oModel:GetModel( cDetail + 'DETAIL' )
    // Obtemos a estrutura de dados do item
    oStruct   := oAux:GetStruct()
    aAux      := oStruct:GetFields()

09/0

    For nI := 1 To Len( aCpoDetail )
        // Incluímos uma linha nova

        // ATENÇÃO: O itens são criados em uma estrutura de grid (FORMGRID),
portanto já é criada uma primeira linha

        //branco automaticamente, desta forma começamos a inserir novas linhas a
partir da 2ª vez
        If nI > 1

            // Incluímos uma nova linha de item
            If ( nItErro := oAux:AddLine() ) <> nI

// Se por algum motivo o método AddLine() não consegue incluir a linha,
// ele retorna a quantidade de linhas já
// existem no grid. Se conseguir retorna a quantidade mais 1

            lRet      := .F.
            Exit
        EndIf
    EndIf

    For nJ := 1 To Len( aCpoDetail[nI] )
// Verifica se os campos passados existem na estrutura de item

```

```

                If ( nPos := aScan( aAux, { |x| AllTrim( x[3] ) == AllTrim(
aCpoDetail[nI][nJ][1] ) } ) ) > 0
                    If !( lAux := oModel:SetValue( cDetail + 'DETAIL',
aCpoDetail[nI][nJ][1], aCpoDetail[nI][nJ][2] ) )
                        // Caso a atribuição não possa ser feita, por algum motivo
                        (validação, por exemplo)
                        // o método SetValue retorna .F.

                        lRet      := .F.
                        nItErro := nI
                        Exit
                    EndIf
                EndIf
            Next
            If !lRet
                Exit
            EndIf
        Next
    EndIf

    If lRet
        // Faz-se a validação dos dados, note que diferentemente das tradicionais "rotinas
        automáticas"
        // neste momento os dados não são gravados, são somente validados.
        If ( lRet := oModel:VldData() )
            // Se o dados foram validados faz-se a gravação efetiva dos
            // dados (commit)
            oModel:CommitData()
        EndIf
    EndIf

    If !lRet
        // Se os dados não foram validados obtemos a descrição do erro para gerar
        // LOG ou mensagem de aviso
        aErro := oModel:GetErrorMessage()
        // A estrutura do vetor com erro é:
        // [1] identificador (ID) do formulário de origem
        // [2] identificador (ID) do campo de origem
        // [3] identificador (ID) do formulário de erro
        // [4] identificador (ID) do campo de erro
        // [5] identificador (ID) do erro
    
```

```

// [6] mensagem do erro
// [7] mensagem da solução
// [8] Valor atribuído
// [9] Valor anterior

AutoGrLog( "Id do formulário de origem:" + ' [' + AllToChar( aErro[1] ) + ']' )
AutoGrLog( "Id do campo de origem:      " + ' [' + AllToChar( aErro[2] ) + ']' )
AutoGrLog( "Id do formulário de erro:  " + ' [' + AllToChar( aErro[3] ) + ']' )
AutoGrLog( "Id do campo de erro:       " + ' [' + AllToChar( aErro[4] ) + ']' )
AutoGrLog( "Id do erro:                " + ' [' + AllToChar( aErro[5] ) + ']' )
AutoGrLog( "Mensagem do erro:         " + ' [' + AllToChar( aErro[6] ) + ']' )
AutoGrLog( "Mensagem da solução:      " + ' [' + AllToChar( aErro[7] ) + ']' )
AutoGrLog( "Valor atribuído:          " + ' [' + AllToChar( aErro[8] ) + ']' )
AutoGrLog( "Valor anterior:           " + ' [' + AllToChar( aErro[9] ) + ']' )
If nItErro > 0

    AutoGrLog( "Erro no Item:          " + ' [' + AllTrim( AllToChar(
nItErro ) ) + ']' )
    EndIf

    MostraErro()

EndIf

// Desativamos o Model

oModel:DeActivate()

Return lRet

```

A situation that can be found when converting an existing application for the MVC structure is the fact that the application is ready to work with the automatic routine so there may be several other applications that use this automatic routine.

The ***FWMVCRotAuto*** function was created so that these applications do not need to change the way they work as today they use the so called standard routine. The application was converted to MVC.

The function uses passed parameters in the prior format of the automatic routine (***MSEXECAUTO***) and performs the model instantiation, attributing values and validation in the MVC format, guaranteeing the legacy programs.

The syntax is:

```
FWMVCRotAuto( oModel, cAlias, nOpcAuto, aAuto, lSeek, lPos )
```

Where:

<b>oModel</b>	Object with the data form model;
<b>cAlias</b>	Main Browse alias;
<b>nOpcAuto</b>	Code of identification of the automatic routine processing type;  [3] Adding  [4] Editing  [5] Deleting
<b>aAuto</b>	Array with the automatic routine in the following structure;  [n][1] Code of the Model form that has an attribution;  [n][2] Standard array of EnchAuto and GetDAuto data, according to the previous documentation;
<b>ISseek</b>	Indicates if the main file must be positioned based on the data provided;
<b>IPos</b>	Indicates if nOpc must not be calculated based on aRotina;

When the application in MVC is converted, it can work in two ways:

- The automatic routine and the
- Model instantiation.

In the following example, we have a registration routine in which there is treatment for this, if the **xRotAuto**, **nOpcAuto** data were passed, it indicates that the application was named by the automatic routine. so we use **FWMVCRotAuto**.

This construction does not prevent the data model (model) to instantiate directly in other applications.

```
Function MATA030_MVC(xRotAuto,nOpcAuto)
Local oMBrowse

If xRotAuto == NIL
    oBrowse := FWMBrowse():New()
    oBrowse:SetAlias('SA1')
    oBrowse:SetDescription("Cadastro de Clientes")
    oBrowse:Activate()
Else
    aRotina := MenuDef()
    FWMVCRotAuto(ModelDef(),"SA1",nOpcAuto,{{"MATA030_SA1",xRotAuto}})
Endif
```

Return NIL

## 16.Entry points in MVC

Entry points are deviations controlled and executed throughout the applications.

After writing an application using *MVC*, pre-established entry points are automatically available.

The idea of entry points for sources developed using the MVC concept and classes is a little different than the applications developed conventionally.

In the conventional sources, we have a **name** for each entry point created, for example, in **MATA010** – Product Registration, we have the entry points: **MT010BRW**, **MTA010OK**, **MT010CAN**, etc. In *MVC*, this is not how it works.

In *MVC*, we create a single entry point and it is called in several moments in the developed application.

This entry point must be a **User Function** and have the identifier (ID) of the data model (Model) of the source as name. Let's take an source of the Legal Module as an example: **JURA001**. In this source, the identifier (*ID*) of the data model (defined in the **ModelDef** function) is also **JURA001**, so when writing the entry point of this application, you must enter:

```
User Function JURA001()  
Local aParam := PARAMIXB  
Local xRet    := .T.  
...  
Return xRet
```

The entry point created received a vector with information regarding the application via parameter (**PARAMIXB**). These parameters vary for each situation. They all have the first three elements that are listed below, in the following board there a list of parameters for each ID:

Arran positions of common parameters for all ID's:

POS.	TYPE	DESCRIPTION
1	O	Form or model object, according to case
2	C	ID of the entry point's place of execution
3	C	Form ID



As previously mentioned, the entry point is called in several moments within the application. In the second position of the vector structure, an identifier (ID) that identifies this moment is transferred. It may have as content:

ENTRY POINT ID	ENTRY POINT'S MOMENT OF EXECUTION
MODELPRE	<p>Before editing any model field.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>
MODELPOS	<p>In the model total validation.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>
FORMPRE	<p>Before editing any form field.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>
FORMPOS	<p>In the form total validation.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>

FORMLINEPRE	<p>Before editing the form row <i>FWFORMGRID</i>.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> <li>4 N <i>FWFORMGRID</i> row number.</li> <li>5 C <i>FWFORMGRID</i> action.</li> <li>6 C Field ID.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>
FORMLINEPOS	<p>In the total validation of the <i>FWFORMGRID</i> form line.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> <li>4 N <i>FWFORMGRID</i> row number.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>
MODELCOMMITTS	<p>After the total saving of the model and in the transaction.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Do not expect return.</p>
MODELCOMMITNTS	<p>After the total saving of the model and in the transaction.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object, as the case may be.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Do not expect return.</p>

FORMCOMMITTS PRE	<p>Before saving the form table.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object according to the case.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> <li>4 L If .T. (true) it indicates the new registration (Adding). If .F. (false), the registration already exists (Editing/Deleting).</li> </ol> <p>Return:</p> <p>Does not expect return.</p>
FORMCOMMITTS POS	<p>After saving the form table.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object according to the case.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> <li>4 L If .T. (true) it indicates the new registration (Adding). If .F. (false), the registration already exists (Editing/Deleting).</li> </ol> <p>Return:</p> <p>Do not expect return.</p>
FORMCANCEL	<p>In the cancellation of the button.</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object according to the case.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>
MODELVLDACTIVE	<p>In the model activation</p> <p>Parameters received:</p> <ol style="list-style-type: none"> <li>1 The form or model object according to the case.</li> <li>2 C ID of the entry point's place of execution.</li> <li>3 C Form ID.</li> </ol> <p>Return:</p> <p>Requires a logical return.</p>

## BUTTONBAR

To add buttons in the ControlBar.

To create the buttons, you must return a two-dimensional array with the following structure of each item:

- |   |   |                             |
|---|---|-----------------------------|
| 1 | C | Title for the button.       |
| 2 | C | Name of Bitmap for display. |
| 3 | B | CodeBlock to be executed.   |
| 4 | C | ToolTip (Optional).         |

Parameters received:

- |   |   |   |
|---|---|---|
| 1 |   | The form or model object according to the case. |
| 2 | C | ID of the entry point's place of execution.     |
| 3 | C | Form ID.  |

Return:

Requires a return array with pre-defines structure.

### Notes:

- When the data model has several components (for example, grid), the third vector position brings the identifier (ID) of this component;
- When the type of return of an execution is not transferred or is transferred with the wrong type, a message is displayed in the console. All ID's that expect return must be treated in the entry point.

### Important:

- When you enter a source in MVC, a **User Function** is created. Be careful when assigning the data model (Model) identifier (ID) as it cannot have the same name as the source (PRW). If the source has the name **FONT001**, the data model (Model) identifier (ID) cannot be **FONT001** as well, as you cannot create another **User Function** named **FONT001** (data model ID) for the entry points.

### Example:

```
User Function JURA001()  
Local aParam      := PARAMIXB  
Local xRet        := .T.  
Local oObj        := ''  
Local cIdPonto    := ''  
Local cIdModel    := ''  
Local lIsGrid     := .F.  
Local nLinha      := 0  
Local nQtdLinhas  := 0  
Local cMsg        := ''  
If aParam <> NIL
```

```

oObj      := aParam[1]
cIdPonto  := aParam[2]
cIdModel  := aParam[3]
lIsGrid   := ( Len( aParam ) > 3 )

    If lIsGrid
        nQtdLinhas := oObj:GetQtdLine()
        nLinha      := oObj:nLine
    EndIf

    If cIdPonto == 'MODELPOS'
        cMsg := 'Chamada na validação total do modelo (MODELPOS).' + CRLF
        cMsg += 'ID ' + cIdModel + CRLF
        If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )
            Help( ,, 'Help',,, 'O MODELPOS retornou .F.', 1, 0 )
        EndIf
    ElseIf cIdPonto == 'FORMPOS'
        cMsg := 'Chamada na validação total do formulário (FORMPOS).' + CRLF
        cMsg += 'ID ' + cIdModel + CRLF
        If cClasse == 'FWFORMGRID'
            cMsg += 'É um FORMGRID com ' + Alltrim( Str( nQtdLinhas ) ) + ;
' linha(s).' + CRLF
        cMsg += 'Posicionado na linha ' + Alltrim( Str( nLinha ) ) + CRLF
        ElseIf cClasse == 'FWFORMFIELD'
            cMsg += 'É um FORMFIELD' + CRLF
        EndIf
        If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )
            Help( ,, 'Help',,, 'O FORMPOS retornou .F.', 1, 0 )
        EndIf
    ElseIf cIdPonto == 'FORMLINEPRE'
        If aParam[5] == 'DELETE'
            cMsg := 'Chamada na pré validação da linha do formulário (FORMLINEPRE).' + CRLF
            cMsg += 'Onde esta se tentando deletar uma linha' + CRLF
            cMsg += 'É um FORMGRID com ' + Alltrim( Str( nQtdLinhas ) ) + ;
' linha(s).' + CRLF
            cMsg += 'Posicionado na linha ' + Alltrim( Str( nLinha ) ) + ; CRLF
            cMsg += 'ID ' + cIdModel + CRLF
            If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )
                Help( ,, 'Help',,, 'O FORMLINEPRE retornou .F.', 1, 0 )
            EndIf
        EndIf
    ElseIf cIdPonto == 'FORMLINEPOS'

```

```

cMsg := 'Chamada na validação da linha do formulário (FORMLINEPOS).' + CRLF
      cMsg += 'ID ' + cIdModel + CRLF
      cMsg += 'É um FORMGRID com ' + Alltrim( Str( nQtdLinhas ) ) + ;
' linha(s).' + CRLF
      cMsg += 'Posicionado na linha ' + Alltrim( Str( nLinha ) ) + CRLF
      If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )
          Help( ,, 'Help',,, 'O FORMLINEPOS retornou .F.', 1, 0 )
      EndIf
      ElseIf cIdPonto == 'MODELCOMMITTTS'
ApMsgInfo('Chamada apos a gravação total do modelo e dentro da transação
(MODELCOMMITTTS).' + CRLF + 'ID ' + cIdModel )
      ElseIf cIdPonto == 'MODELCOMMITNTTS'
ApMsgInfo('Chamada apos a gravação total do modelo e fora da transação
(MODELCOMMITNTTS).' + CRLF + 'ID ' + cIdModel)
      //ElseIf cIdPonto == 'FORMCOMMITTTSPRE'
      ElseIf cIdPonto == 'FORMCOMMITTTSPOS'
ApMsgInfo('Chamada apos a gravação da tabela do formulário (FORMCOMMITTTSPOS).' + CRLF +
'ID ' + cIdModel)
      ElseIf cIdPonto == 'MODELCANCEL'
cMsg := 'Chamada no Botão Cancelar (MODELCANCEL).' + CRLF + 'Deseja Realmente Sair ?'

      If !( xRet := ApMsgYesNo( cMsg ) )
          Help( ,, 'Help',,, 'O MODELCANCEL retornou .F.', 1, 0 )
      EndIf
      ElseIf cIdPonto == 'MODELVLDACTIVE'
cMsg := 'Chamada na validação da ativação do Model.' + CRLF + ;
'Continua ?'
      If !( xRet := ApMsgYesNo( cMsg ) )
          Help( ,, 'Help',,, 'O MODELVLDACTIVE retornou .F.', 1, 0 )
      EndIf
      ElseIf cIdPonto == 'BUTTONBAR'
ApMsgInfo('Adicionando Botão na Barra de Botões (BUTTONBAR).' + CRLF + 'ID ' + cIdModel )
xRet := { {'Salvar', 'SALVAR', { || Alert( 'Salvou' ) }, 'Este botão Salva' } }
      EndIf
      EndIf
      Return xRet

```

## 17. Web Services for MVC

When you develop an application using *MVC*, a Web Service will already be available to be used for the data receiving.

All applications in *MVC* use the same Web Service, regardless of the structure or how many entities it has.

The Web Service that is available for *MVC* is **FWWSMODEL**.

The basic idea is that we instantiate the Web Service, enter the application to be used and enter the data in an XML format.

### 17.1 Web Service for data models that have an entity

We will learn how to build an application that uses the **FWWSMODEL** Web Service with a data model (Model) that has only one entity.

### 17.2 Instantiation of Web Service Client

The instantiation goes as follows:

Instantiation of Web Service Client.

```
oMVCWS := WsFwWsModel():New()
```

Definition of the **FWWSMODEL** URL in the Web Services server.

```
oMVCWS:_URL := http://127.0.0.1:8080/ws/FWWSMODEL.apw
```

Definition of the application to be used.

We define the source name that has **ModelDef** that we wish to use.

```
oMVCWS:cModelId := 'COMP011_MVC'
```

### 17.3 The XML structure used

As previously said, the data is informed in an XML. The structure of this XML follows this hierarchy:

```
<ID do Model>
  <ID de Componente>
    <ID de Campo>
      Conteúdo...
    </ID de Campo>
  </ID de Componente >
</ID do Model>
```

The **<ID do Model>** tag is the identifier (ID) was defined in the data model (Model) of the *MVC* application.

**Example:**

In the application, we have the following defined:

```
oModel := MPFormModel():New('COMP011M' )
```

In the XML, the **<ID do Model>** tags are:

```
<COMP011M>
    ...
</COMP011M>
```

The **adding (3)**, **editing (4)** or **deleting (5)** operation must also be informed in this tag, in the **Operation** attribute.

So if we want to perform an inclusion operation, we have:

```
<COMP011M Operation="3">
```

The **<Component ID>** tags are ID's of form components or grid components that were defined in the data model (Model) of the application.

**Example:**

If in the application we have:

```
oModel:AddFields( 'ZA0MASTER' )
```

In the XML, the **<Component ID>** tags are:

```
<ZA0MASTER>
    ...
</ZA0MASTER>
```

The component type (form or grid) must also be entered in this tag in the **modeltype** attribute. Enter **FIELDS** for form and **GRID** component for grid components.

We would then have:

```
<ZA0MASTER modeltype="FIELDS">
```

The **<Field ID>** tags are the names of the structure fields of the component, whether form or grid.

If we have the **ZA0\_FILIAL**, **ZA0\_CODIGO** and **ZA0\_NOME** fields, for example, we have:

```
<ZA0_FILIAL>
    ...
</ZA0_FILIAL>

<ZA0_CODIGO>
    ...
</ZA0_CODIGO>

<ZA0_NOME>
```



```
...
</ZA0_NOME>
```

The order of the fields must be entered in these tags, in the **order** attribute.

```
<ZA0_FILIAL order="1">
...
</ZA0_FILIAL>
<ZA0_CODIGO order="2">
...
</ZA0_CODIGO >
<ZA0_NOME order="3">
...
</ZA0_NOME>
```

When the component is a form (**FIELDS**), the data itself must be informed in a **value** tag.

```
<ZA0_FILIAL order="1">
    <value>01</value>
</ZA0_FILIAL>
<ZA0_CODIGO order="2">
    <value>001000</value>
</ZA0_CODIGO >
<ZA0_NOME order="3">
    <value>Tom Jobim</value>
</ZA0_NOME>
```

Então a estrutura completa será:

```
<COMP011M Operation="1">
    <ZA0MASTER modeltype="FIELDS" >
        <ZA0_FILIAL order="1">
            <value>01</value>
        </ZA0_FILIAL>
        <ZA0_CODIGO order="2">
            <value>01000</value>
        </ZA0_CODIGO>
        <ZA0_NOME order="3">
            <value>Tom Jobim</value>
        </ZA0_NOME>
    </ZA0MASTER>
</COMP011M>
```

## 17.4 Obtaining an XML structure of a data model (GetXMLData)

We can obtain an XML structure that an MVC application expects by using the **GetXMLData**

method of the *Web Service*.

**Example:**

```
oMVCWS:GetXMLData()
```

The expected XML is entered in the WS **cGetXMLDataResult** attribute.

```
cXMLEstrut := oMVCWS:cGetXMLDataResult
```

Still using the example above, we have:

```
<?xml version="1.0" encoding="UTF-8"?>
<COMP011M Operation="1"
  <ZA0MASTER modeltype="FIELDS" >
    <ZA0_FILIAL order="1"><value></value></ZA0_FILIAL>
    <ZA0_CODIGO order="2"><value></value></ZA0_CODIGO>
    <ZA0_NOME order="3"><value></value></ZA0_NOME>
  </ZA0MASTER>
</COMP011M>
```

## 17.5 Entering the XML data to the Web Service

The XML including the data must be attributed to the **cXML** attribute of the *Web Service* object.

**Example:**

```
oMVCWS:cXML := cXML // variável que contém o XML com os dados
```

## 17.6 Validating the data (VldXMLData)

To submit these data to the data model (Model) to be validated, we use the **VldXMLData** method.

```
If !oMVCWS:VldXMLData()
  MsgStop( 'Problemas na validação dos dados' + CRLF + WSError() )
EndIf
```

In this moment, the data is validated but not saved. **VldXMLData** only validates.

This is an interesting resource if we wish to perform a simulation, for example.

## 17.7 Validating and saving data (PutXMLData)

The difference between the **VldXMLData** method and the **PutXMLData** method is that the **PutXMLData** besides submitting the XML data to the data model for validation, also saves this data if the validation is successful.

The result is informed in the **IPutXMLDataResult** attribute and if there is some problem, it is described in the **cVldXMLDataResult** attribute of the Web Service object.

```
If oMVCWS:PutXMLData()  
    If oMVCWS:lPutXMLDataResult  
        MsgInfo( 'Informação gravada com sucesso.' )  
    Else  
        MsgStop( 'Informação não gravada ' + CRLF + WSError() )  
    EndIf  
Else  
    MsgStop( AllTrim(oMVCWS:cVldXMLDataResult) + CRLF + WSError() )  
EndIf
```

## 17.8 Obtaining the XSD scheme of a data model (GetSchema)

The XML informed before the validation of information of the data model (Model) is validated by XSD *schema* regarding the model. This validation is made automatically and XSD is based in the data model (Model) structure.

This validation refers to the XML structuring (tags, levels, orders, etc.) and not to XML data, the data validation is the business rule function.

If the developer wants to obtain the XSD *schema* to be used, the **GetSchema** method may be used.

### Example:

```
If oMVCWS:GetSchema()  
    cXMLEsquema := oMVCWS:cGetSchemaResult  
EndIf
```

The XSD *schema* is returned in the cGetSchemaResult attribute of the *Web Service* object.

## 17.9 Complete example of Web Service

```
User Function COMPW011()

Local oMVCWS

// Instancia o WebService Genérico para Rotinas em MVC
oMVCWS := WsFwWsModel():New()

// URL onde esta o WebService FWWSModel do Protheus
oMVCWS:_URL      := http://127.0.0.1:8080/ws/FWWSMODEL.apw

// Seta Atributos do WebService
oMVCWS:cModelId  := 'COMP011_MVC' // Fonte de onde se usara o Model

// Exemplo de como pegar a descrição do Modelo de Dados
//If oMVCWS:GetDescription()
//    MsgInfo( oMVCWS:cGetDescriptionResult )
//Else
//    MsgStop( 'Problemas em obter descrição do Model' + CRLF + WSError() )
//EndIf

// Obtém a estrutura dos dados do Model
If oMVCWS:GetXMLData()

    // Retorno da GetXMLData
    cXMLEstrut := oMVCWS:cGetXMLDataResult
    // Retorna
    //<?xml version="1.0" encoding="UTF-8"?>
    //<COMP011M Operation="1" version="1.01">
    //    <ZA0MASTER modeltype="FIELDS" >
    //        <ZA0_FILIAL order="1"><value></value></ZA0_FILIAL>
    //        <ZA0_CODIGO order="2"><value></value></ZA0_CODIGO>
    //        <ZA0_NOME order="3"><value></value></ZA0_NOME>
    //    </ZA0MASTER>
    //</COMP011M>

    // Obtém o esquema de dados XML (XSD)
    If oMVCWS:GetSchema()
        cXMLEschema := oMVCWS:cGetSchemaResult
    EndIf

    // Cria o XML
    cXML := '<?xml version="1.0" encoding="UTF-8"?>'
    cXML += '<COMP011M Operation="1" version="1.01">'
    cXML += '    <ZA0MASTER modeltype="FIELDS" >'
```

```

cXML += '          <ZA0_FILIAL order="1"><value>01</value></ZA0_FILIAL>'
cXML += '          <ZA0_CODIGO order="2"><value>000100</value></ZA0_CODIGO>'
cXML += '          <ZA0_NOME   order="3"><value>Tom Jobim</value></ZA0_NOME>'
cXML += '        </ZA0MASTER>'
cXML += '</COMP011M>'

// Joga o XML para o atributo do Webservice
oMVCWS:cModelXML := cXML

// Valida e Grava os dados
If oMVCWS:PutXMLData()
    If oMVCWS:lPutXMLDataResult
        MsgInfo( 'Informação Importada com sucesso.' )
    Else
        MsgStop( 'Não importado' + CRLF + WSError() )
    EndIf
Else
    MsgStop( AllTrim( oMVCWS:cVldXMLDataResult ) + CRLF + WSError() )

EndIf

```

## 17.10 Web Services for data models with two or more entities

For construction of *Web Services* with two or more entities, what is different is only the XML received that has more levels. Notice the example of the source:

```

#include 'PROTHEUS.CH'
#include 'XMLXFUN.CH'
#include 'FWMVCDEF.CH'

//-----
/*{Protheus.doc} COMPW021
Exemplo de utilizacao do Webservice generico para rotinas em MVC
para uma estrutura de pai/filho
@author Ernani Forastieri e Rodrigo Antonio Godinho
@since 05/10/2009
@version P10
*/
//-----

User Function COMPW021()
Local oMVCWS

```

```

Local cXMLEstrut  := ''
Local cXMLESquema := ''
Local cXMLFile    := '\XML\WSMVCTST.XML'

RpcSetType( 3 )
RpcSetEnv( '99', '01' )

// Instancia o Webservice Generico para Rotinas em MVC
oMVCWS := WsFwWsModel():New()
oMVCWS:_URL      := "http://127.0.0.1:8080/ws/FWWSMODEL.apw"
oMVCWS:cUserLogin := 'admin'
oMVCWS:cUserToken := 'admin'
oMVCWS:cPassword  := ''
oMVCWS:cModelId   := 'COMP021_MVC' // Fonte de onde se usara o Model

// Obtem a estrutura dos dados do Model
If oMVCWS:GetXMLData()
    If oMVCWS:GetSchema()
        cXMLESquema := oMVCWS:cGetSchemaResult
    EndIf

    cXMLEstrut := oMVCWS:cGetXMLDataResult

    <?xml version="1.0" encoding="UTF-8"?>
    <COMP021MODEL Operation="1" version="1.01">
    <ZA1MASTER modeltype="FIELDS" >
    <ZA1_FILIAL order="1"><value></value></ZA1_FILIAL>
    <ZA1_MUSICA order="2"><value></value></ZA1_MUSICA>
    <ZA1_TITULO order="3"><value></value></ZA1_TITULO>
    <ZA1_DATA order="4"><value></value></ZA1_DATA>
    <ZA2DETAIL modeltype="GRID" >
    <struct>
    <ZA2_FILIAL order="1"></ZA2_FILIAL>
    <ZA2_MUSICA order="2"></ZA2_MUSICA>
    <ZA2_ITEM order="3"></ZA2_ITEM>
    <ZA2_AUTOR order="4"></ZA2_AUTOR>
    </struct>
    <items>
    <item id="1" deleted="0" >
    <ZA2_FILIAL></ZA2_FILIAL>

```

```

//          <ZA2_MUSICA></ZA2_MUSICA>
//          <ZA2_ITEM></ZA2_ITEM>
//          <ZA2_AUTOR></ZA2_AUTOR>
//      </item>
//  </items>
//  </ZA2DETAIL>
//</ZA1MASTER>
//</COMP021MODEL>
// Obtem o esquema de dados XML (XSD)
If oMVCWS:GetSchema()
    cXMLEsquema := oMVCWS:cGetSchemaResult
EndIf

cXML := ''
cXML += '<?xml version="1.0" encoding="UTF-8"?>'
cXML += '<COMP021MODEL Operation="1" version="1.01">'
cXML += '<ZA1MASTER modeltype="FIELDS">'
cXML += '<ZA1_FILIAL order="1"><value>01</value></ZA1_FILIAL>'
cXML += '<ZA1_MUSICA order="2"><value>000001</value></ZA1_MUSICA>'
cXML += '<ZA1_TITULO order="3"><value>AQUARELA</value></ZA1_TITULO>'
cXML += '<ZA1_DATA order="4"><value></value></ZA1_DATA>'
cXML += '    <ZA2DETAIL modeltype="GRID" >'
cXML += '        <struct>'
cXML += '            <ZA2_FILIAL order="1"></ZA2_FILIAL>'
cXML += '            <ZA2_MUSICA order="2"></ZA2_MUSICA>'
cXML += '            <ZA2_ITEM order="3"></ZA2_ITEM>'
cXML += '            <ZA2_AUTOR order="4"></ZA2_AUTOR>'
cXML += '        </struct>'
cXML += '        <items>'
cXML += '            <item id="1" deleted="0" >'
cXML += '                <ZA2_FILIAL>01</ZA2_FILIAL>'
cXML += '                <ZA2_MUSICA>000001</ZA2_MUSICA>'
cXML += '                <ZA2_ITEM>01</ZA2_ITEM>'
cXML += '                <ZA2_AUTOR>000001</ZA2_AUTOR>'
cXML += '            </item>'
cXML += '            <item id="2" deleted="0" >'
cXML += '                <ZA2_FILIAL>01</ZA2_FILIAL>'
cXML += '                <ZA2_MUSICA>000002</ZA2_MUSICA>'
cXML += '                <ZA2_ITEM>02</ZA2_ITEM>'
cXML += '                <ZA2_AUTOR>000002</ZA2_AUTOR>'

```

```

cXML += '                </item>'
cXML += '            </items>'
cXML += '    </ZA2DETAIL>'
cXML += '</ZA1MASTER>'
cXML += '</COMP021MODEL>'

// Joga o XML para o atributo do Webservice
oMVCWS:cModelXML := cXML

// Valida e Grava os dados
If oMVCWS:PutXMLData()
    If oMVCWS:lPutXMLDataResult
        MsgInfo( 'Informação importada com sucesso.' )
    Else
        MsgStop( 'Não importado' + CRLF + WSError() )
    EndIf
Else
    MsgStop( AllTrim( oMVCWS:cVldXMLDataResult ) + CRLF + WSError() )
EndIf
Else
    MsgStop( 'Problemas em obter Folha de Dados do Model' + CRLF + WSError() )
EndIf
RpcClearEnv()

Return NIL

//-----
Static Function WSError()

Return IIf( Empty( GetWscError(3) ), GetWscError(1), GetWscError(3) )

```



## 18. Use of New Model command

To make the development easier, commands were created to generate an application in MVC in a simple and fast way. It is the **New Model** command.

This command is indicated for those applications with a table (old **Model1**) or a non-normalized table (header and item in the same registration) being used, but with the need to work in a **master-detail** structure (old **Model2**) or where two tables are used in a **master-detail** structure (old **Model3**).

Using the **New Model** command, you must not enter all the functions and classes usually used in an MVC routine. During the pre-compilation process, the **New Model** and its directives are changed into an MVC source that uses **FWmBrowse**, **ModelDef**, **ViewDef** and occasionally **MenuDef**.

The premise to use this command is one of the constructions mentioned above and the structures of the tables are defined in the SX3 dictionaries. Structures cannot be manually constructed or if you add or remove structure fields.

As this command is a compilation policy of **#COMMAND** type, to use this command you must add the following policy in the source:

```
#INCLUDE 'FWMVCDEF.CH'
```

Below is the syntax of the command and examples of use.

### 18.1 New Model Syntax

This is the **New Model** command syntax:

#### NEW MODEL

```

TYPE                <nType> ;
DESCRIPTION         <cDescription> ;
BROWSE              <oBrowse> ;
SOURCE              <cSource> ;
MODELID             <cModelID> ;
FILTER               <cFilter> ;

CANACTIVE <bSetVldActive> ;
PRIMARYKEY <aPrimaryKey> ;
MASTER <cMasterAlias> ;
HEADER <aHeader,...> ;
BEFORE <bBeforeModel> ;

```

**AFTER** <*bAfterModel*> ;  
**COMMIT** <*bCommit*> ;  
**CANCEL** <*bCancel*> ;  
**BEFOREFIELD** <*bBeforeField*> ;  
**AFTERFIELD** <*bAfterField*> ;  
**LOAD** <*bFieldLoad*> ;  
     **DETAIL** <*cDetailAlias*> ;  
     **BEFORELINE** <*bBeforeLine*> ;  
     **AFTERLINE** <*bAfterLine*> ;  
     **BEFOREGRID** <*bBeforeGrid*> ;  
     **AFTERGRID** <*bAfterGrid*> ;  
     **LOADGRID** <*bGridLoad*> ;  
     **RELATION** <*aRelation*> ;  
     **ORDERKEY** <*cOrder*> ;  
     **UNIQUELINE** <*aUniqueLine*> ;  
     **AUTOINCREMENT** <*cFieldInc*> ;  
**OPTIONAL**

Where:

**TYPE** <*nType*>

Numeric Type - Mandatory

Structure Type

1 = 1 Table

2 = 1 Table Master/Detail

3 = 2 Tables Master/Detail

**DESCRIPTION** <cDescription>

Character Type - Mandatory

Routine Description

**BROWSE** <oBrowse>

Object Type - Mandatory

Browse Object to be used

**SOURCE** <cSource>

Character Type - Mandatory

Source Name

**MODELID** <cModelID>

Character Type - Mandatory

Model identifier (ID).

**FILTER** <cFilter>

Character Type - Optional

Browse Filter

**CANACTIVE** <bSetVldActive>

Block Type - Optional

Block for validation of Model activation. It receives Model as parameter

Ex. { |oModel| COMP011ACT( oModel ) }

**PRIMARYKEY** <aPrimaryKey>

Array Type - Optional

Array with primary Browse keys, if not informed, it searches the table's X2\_UNICO.

**MASTER** <cMasterAlias>

Character Type - Mandatory

Main Table (Master)

**HEADER** <aHeader>

Array Type - Mandatory for TYPE = 2

Array with fields to be considered in the Header

**BEFORE** <bBeforeModel>

Block Type - Optional

Model Pre-Validation Block. It receives the Model as parameter.

Ex. { |oModel| COMP011PRE( oModel ) }

**AFTER** <bAfterModel>

Block Type - Optional

Block of Model Post-Validation. It receives the Model as parameter.

Ex. { |oModel| COMP011POS( oModel ) }

**COMMIT** <bCommit>

Block Type - Optional

Block of persistence of the data (Commit) of the Model. It receives the Model as parameters.

Ex. { |oModel| COMP022CM( oModel ) }

**CANCEL** <bCancel>

Block Type - Optional

Block enabled in the Cancel button. It receives the Model as parameters.

Ex. { |oModel| COMP011CAN( oModel ) }

**BEFOREFIELD** <bBeforeField>

Block Type - Optional

Block of Pre-Validation of FORMFIELD of the Master table. It receives the ModelField, the identifier (ID) of the place of execution and the form identifier (ID) as parameter

Ex. { |oMdlF,cId ,cidForm| COMP023FPRE( oMdlF,cId ,cidForm) }

**AFTERFIELD** <bAfterField>

Block Type - Optional

Block of Post Validation of FORMFIELD of the Master table. . It receives the ModelField, the identifier (ID) of the place of execution and the form identifier (ID) as parameter

Ex. { |oMdlF,cId ,cidForm| COMP023FPOS( oMdlF,cId ,cidForm) }

**LOAD** <bFieldLoad>

Block Type - Optional

Block of data Load of FORMFIELD of the Master field

**DETAIL** <cDetailAlias>

Character type - Mandatory for TYPE = 2 or 3

Detail Table

**BEFORELINE** <bBeforeLine>

Block Type - Optional

Block of Post-Validation of the FORMGRID row of the Detail table. It receives the ModelGrid, the FORMGRID row number, the action and the FORMGRID field as parameters.

Ex. { |oMdIG,nLine,cAcao,cCampo| COMP023LPRE( oMdIG, nLine, cAcao, cCampo ) }

Used only for TYPE = 2 or 3

**AFTERLINE** <bAfterLine>

Block Type - Optional

Block of Post-Validation of the FORMGRID row of the Detail table. It receives the ModelGrid and the FORMGRID row number as parameter.

Ex. { |oModelGrid, nLine| COMP022LPOS( oModelGrid, nLine ) }

Used only for TYPE = 2 or 3

**BEFOREGRID** <bBeforeGrid>

Block Type - Optional

Block of Pre-Validation of FORMGRID of the Detail table. It receives the ModelGrid as parameter.

Used only for TYPE = 2 or 3

**AFTERGRID** <bAfterGrid>

Block Type - Optional

Block of Pre-Validation of FORMGRID of the Detail table. It receives the ModelGrid as parameter.

Used only for TYPE = 2 or 3

**LOADGRID** <bGridLoad>

Block Type - Optional

Block of data load of FORMGRID of the Detail table

Used only for TYPE = 2 or 3

**RELATION** <aRelation>

Array Type - Mandatory for TYPE = 2 or 3

Two-dimensional array for relationship with Master/Detail tables.

Used only for TYPE = 2 or 3

**ORDERKEY** <cOrder>

Character Type - Optional

Order of FORMGRID of Detail table

Used only for TYPE = 2 or 3

**UNIQUELINE** <aUniqueLine>

Array Type - Optional

Array with fields that can be duplicated in FORMGRID of the Detail table

Used only for TYPE = 2 or 3

**AUTOINCREMENT** <cFieldInc>

Array Type - Optional

Auto incrementing fields for FORMGRID of the Detail table

Used only for TYPE = 2 or 3

**OPTIONAL**

It indicates if the completion of the FORMGRID of the Detail table is optional

Used only for TYPE = 2 or 3

**Example:**

```
//
// Construcao para uma tabela
//

#include "PROTHEUS.CH"
#include "FWMVCDEF.CH"

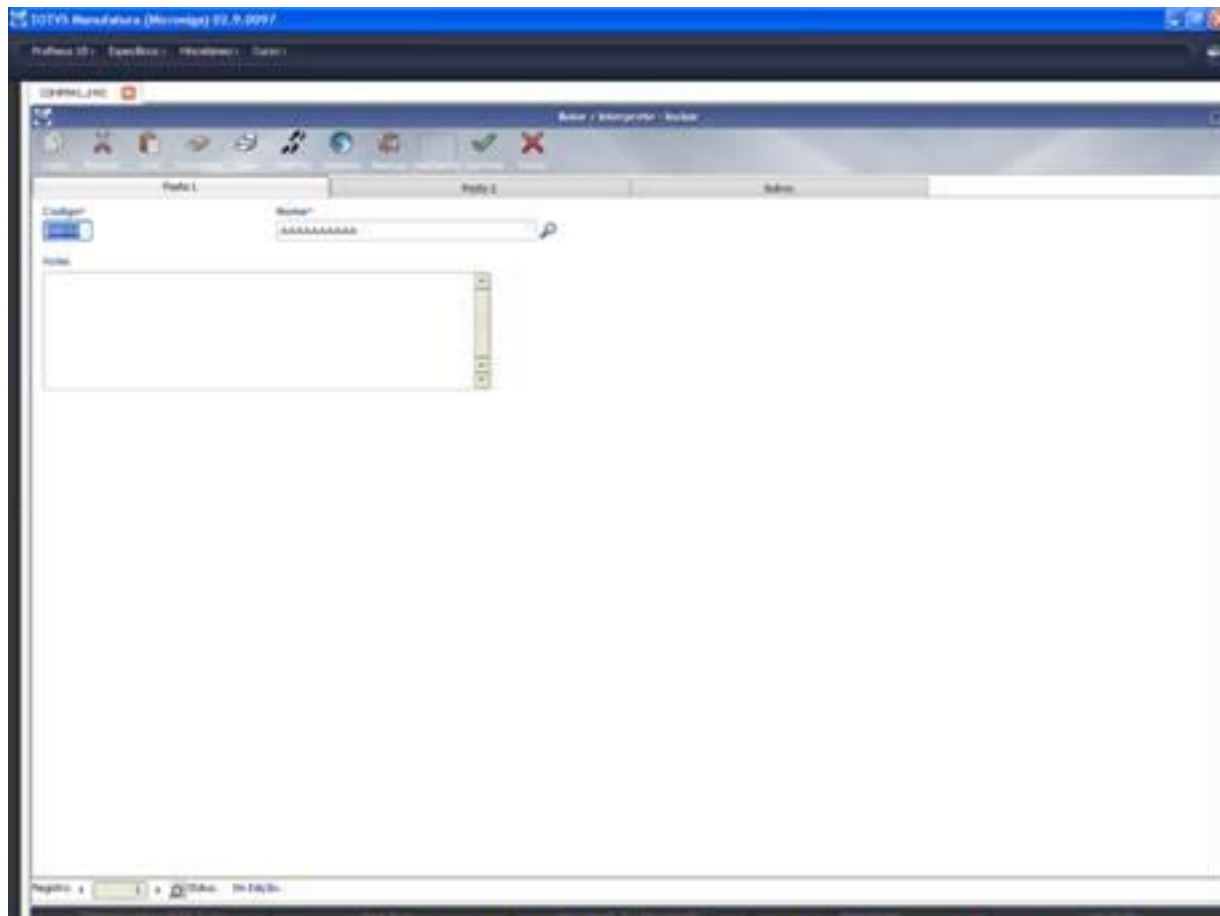
User Function COMP041_MVC()
Local oBrowse

NEW MODEL ;
TYPE      1 ;
DESCRIPTION "Cadastro de Autor/Interprete" ;
BROWSE     oBrowse      ;
SOURCE     "COMP041_MVC" ;
MODELID    "MDCOMP041"   ;
FILTER     "ZA0_TIPO=='1'" ;
MASTER     "ZA0"         ;
AFTER      { |oMdl| COMP041POS( oMdl ) } ;
COMMIT     { |oMdl| COMP041CMM( oMdl ) }
Return NIL

Static Function COMP041POS( oModel )
Help( ,, 'Help',, 'Acionou a COMP041POS', 1, 0 )
Return .T.

Static Function COMP041CMM( oModel )
FWFormCommit( oModel )
Return NIL
```

Visually we have:



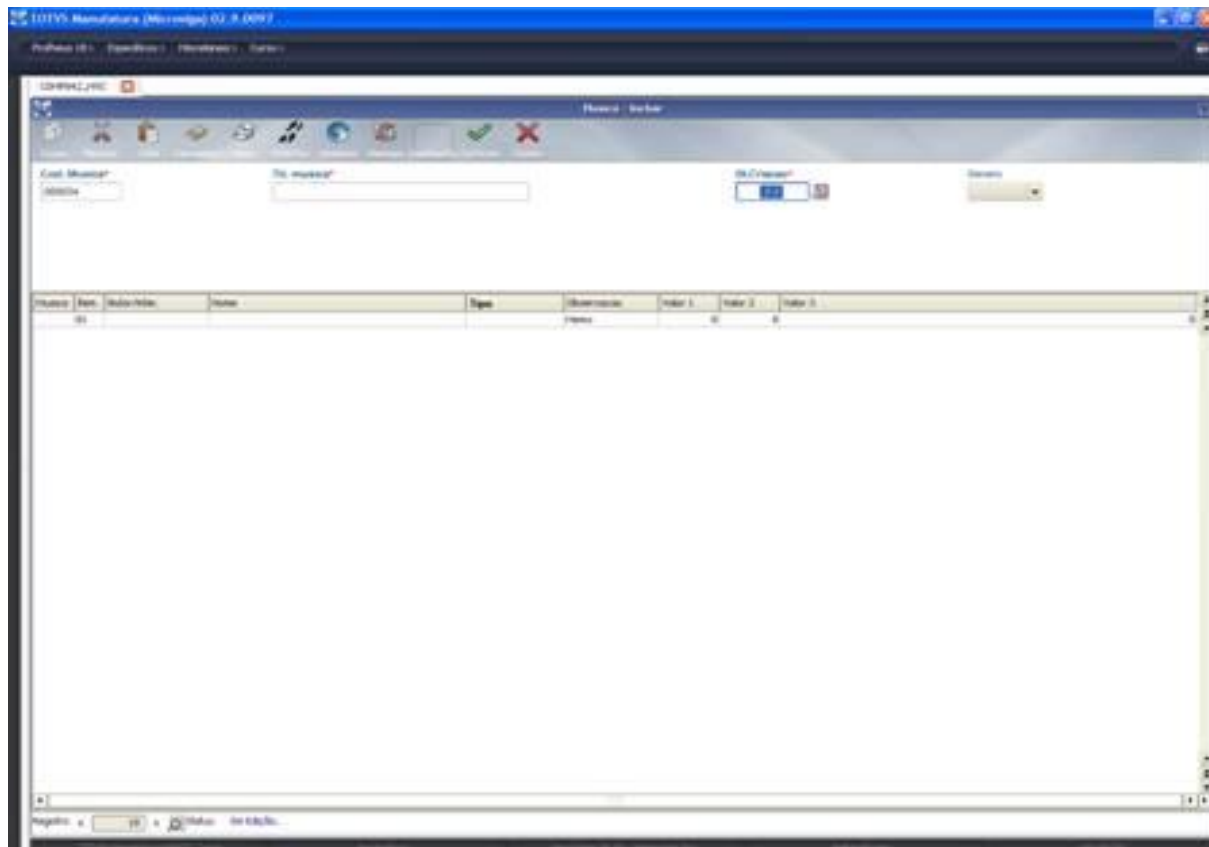
```
//  
// Construcao para uma tabela Master/Detail  
//  
#INCLUDE "PROTHEUS.CH"  
#INCLUDE "FWMVCDEF.CH"  
  
User Function COMP042_MVC()  
Local oBrowse  
  
NEW MODEL ;  
TYPE          2 ;  
DESCRIPTION    "Tabela Nao Normalizada" ;  
BROWSE         oBrowse ;  
SOURCE         "COMP042_MVC" ;  
MODELID        "MDCOMP042" ;  
MASTER        "ZA2" ;
```



Este manual é de propriedade da TOTVS. Todos os direitos reservados.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.



```
//
// Construção para duas tabelas Master/Detail
//
#include "PROTHEUS.CH"
#include "FWMVCDEF.CH"

User Function COMP043_MVC()
Local oBrowse

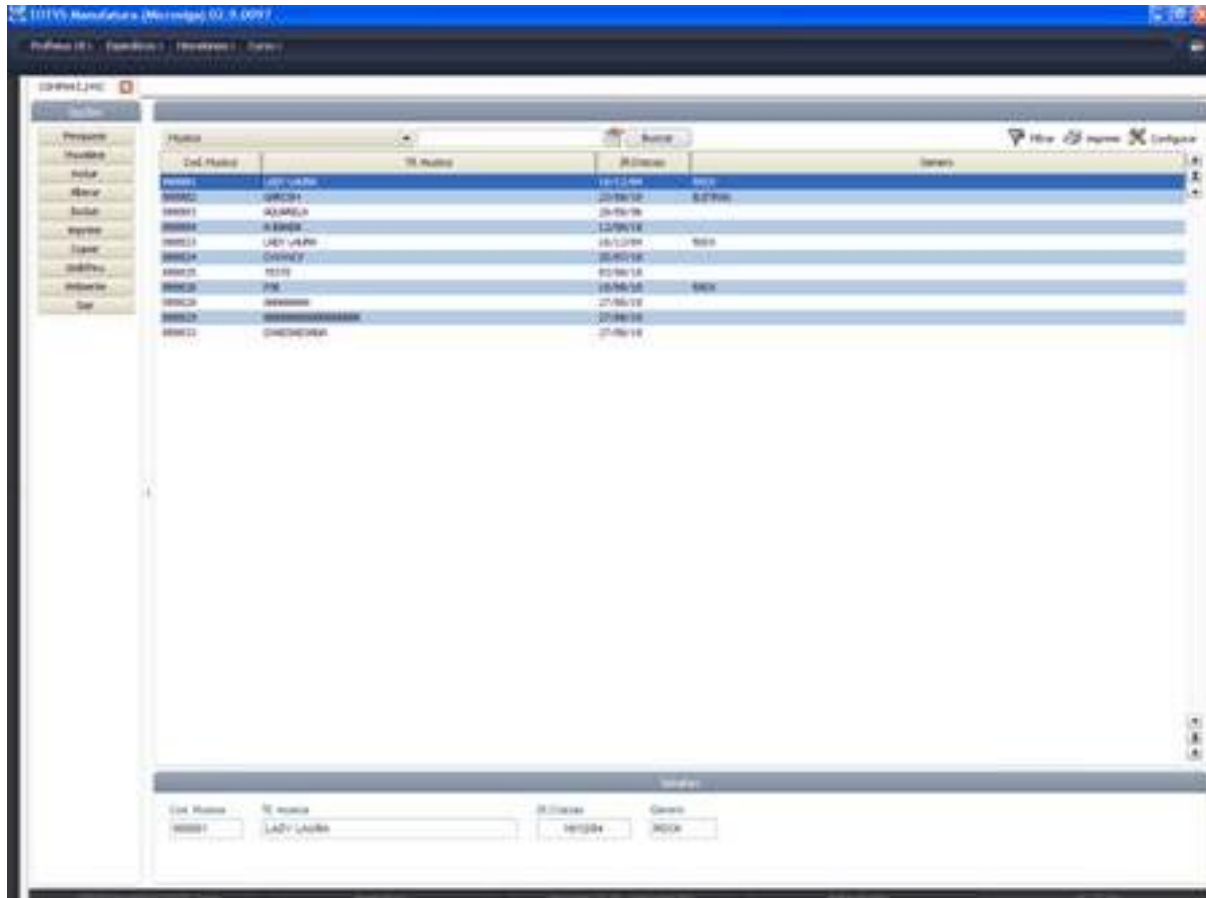
NEW MODEL ;
TYPE      3      ;
DESCRIPTION  "Musicas"      ;
BROWSE      oBrowse      ;
SOURCE      "COMP043_MVC"  ;
MODELID     "MDCOMP043"   ;
MASTER      "ZA1"         ;
DETAIL      "ZA2"         ;
RELATION     { { 'ZA2_FILIAL', 'xFilial( "ZA2" )' }, ;
{ 'ZA2_MUSICA', 'ZA1_MUSICA' } } ;
UNIQUELINE   { 'ZA2_AUTOR' } ;
```

```
ORDERKEY      ZA2->( IndexKey( 1 ) ) ;
```

```
AUTOINCREMENT  'ZA2_ITEM'
```

Return NIL

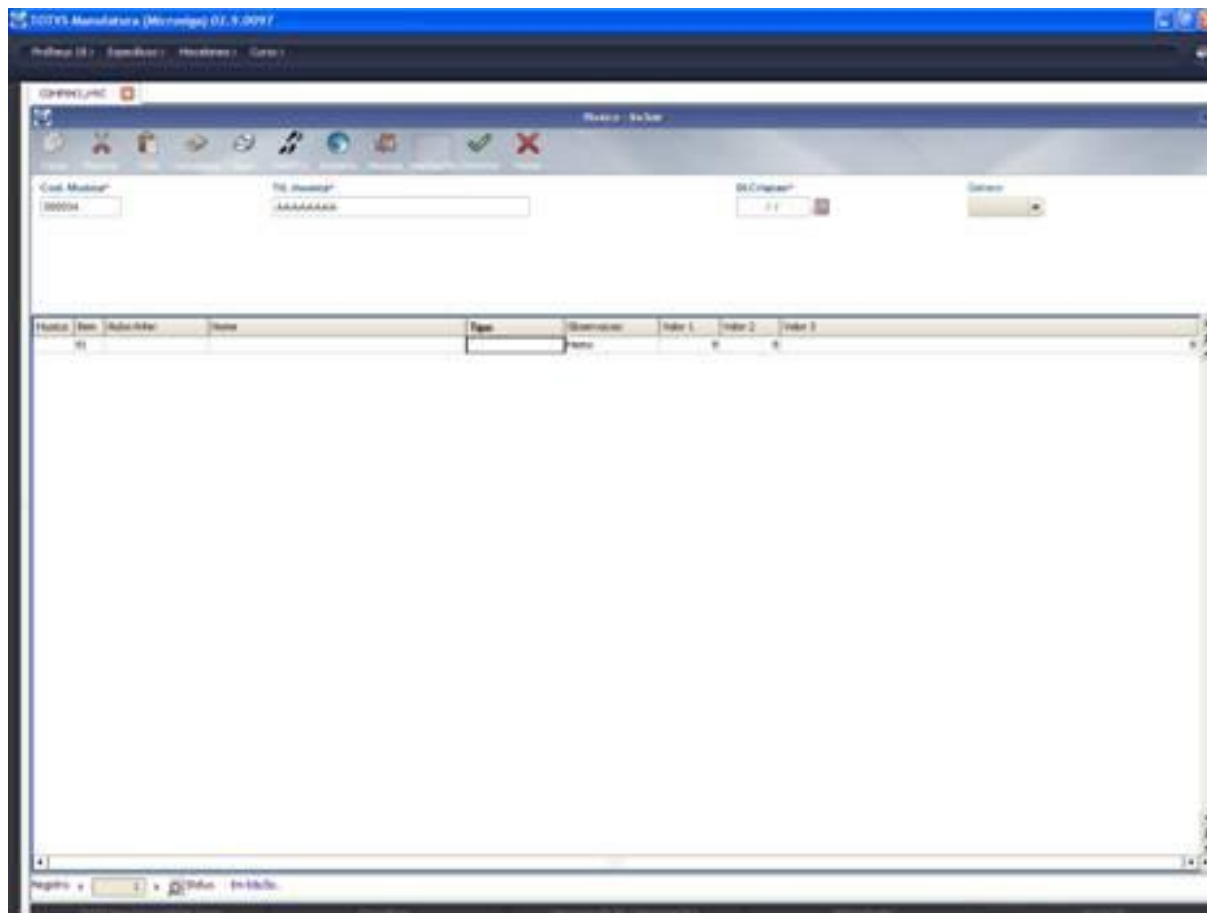
The result is:



Del.Número	Itm.Número	R.Díctame	Genero
000001	LADY LADIES	10/10/18	NOVO
000002	APRIL	20/09/18	6.27M
000003	APRIL	20/09/18	
000004	A. KINGS	12/09/18	
000011	LADY LADIES	10/10/18	NOVO
000014	CHERRY	20/09/18	
000015	10/10	20/09/18	
000016	10/10	20/09/18	NOVO
000018	APRIL	20/09/18	
000019	APRIL	20/09/18	
000021	CHERRY	20/09/18	

Validar

Del.Número: 000001 Itm.Número: LADY LADIES R.Díctame: 10/10/18 Genero: NOVO



```
//
// Construcao para uma tabela com menudef diferenciado
//
#include "PROTHEUS.CH"
#include "FWMVCDEF.CH"

User Function COMP044_MVC()
Local oBrowse

NEW MODEL ;
TYPE      1 ;
DESCRIPTION "Cadastro de Autor/Interprete" ;
BROWSE    oBrowse ;
SOURCE    "COMP044_MVC" ;
MENUEDEF  "COMP044_MVC" ;
MODELID   "MDCOMP044" ;
FILTER    "ZA0_TIPO=='1'" ;
MASTER    "ZA0"
```

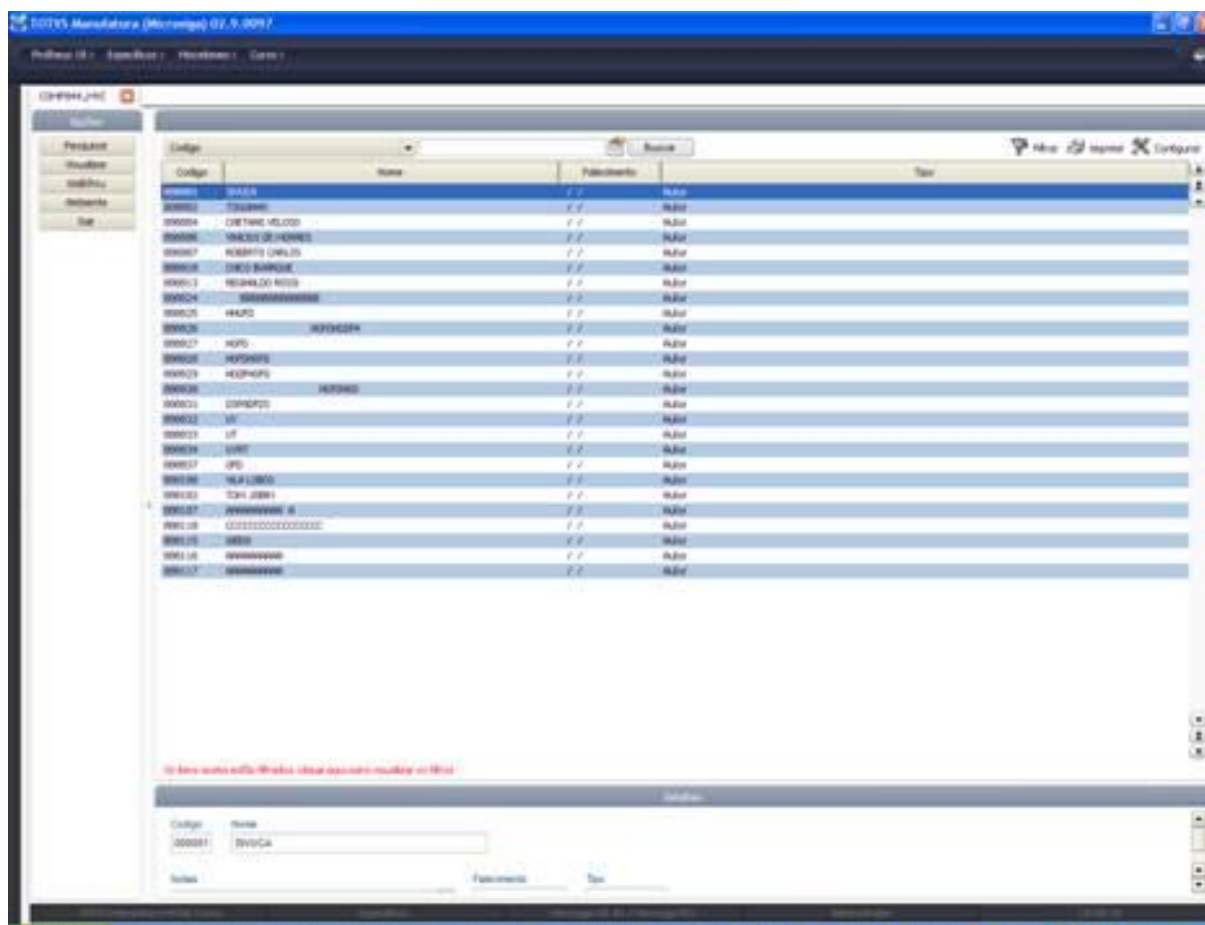
///-----

```
Local aRotina := {}
```

```
ADD OPTION aRotina TITLE 'Visualizar' ACTION 'VIEWDEF.COMP044 MVC' OPERATION 2 ACCESS 0
```

Return aRotina

The result is:



## 19.Reusing the existing data model or interface

One of the great advantages in the construction of applications in MVC is the possibility to reuse the data model (Model) or the interface (View) in other applications, by using the concept of inheritance.

We can reuse the components as they are defined. We can also add new entities to them.

To do this, we must instantiate the data model (Model) or the interface (View) in the new application.

The exemplify this use below.

### 19.1 We only reuse the components

In this example, we carry out the data model (Model) and the interface (View) that exist in an application for the construction of a new one with no editing.

We will use the following functions:

**FWLoadModel**, see chapter 0 12.4 Load the data model of an existing application (FWLoadModel)) and;

**FWLoadView**, see chapter 0 12.5 Load the interface of an existing application (FWLoadView).

In **ModelDef** of the new application, we instanciate the data model (Model) of the existing application:

```
Static Function ModelDef()  
  
Local oModel := FWLoadModel( "COMP011_MVC" )  
  
Return oModel
```

In **MenuDef** of the application, we instantiate the interface (View) of another application:

```
Static Function ModelDef()  
  
Local oModel := FWViewModel( "COMP011_MVC" )  
  
Return oModel
```

In the examples above, the new application uses the same components of the already existing application, in in this case, which is defined in the **ModelDef** of the **COMP011\_MVC** source.

#### Example:

```
#INCLUDE 'PROTHEUS.CH'  
#INCLUDE 'FWMVCDEF.CH'  
  
//-----  
  
User Function COMP015_MVC()  
  
Local oBrowse
```

```

oBrowse := FWMBrowse():New()
oBrowse:SetAlias('ZA0')
oBrowse:SetDescription('Cadastro de Autor/Interprete')
oBrowse:DisableDetails()

oBrowse:Activate()

Return NIL

//-----
Static Function MenuDef()
Return FWLoadMenuDef( "COMP011_MVC")

//-----
Static Function ModelDef()
// Criamos o modelo de dados desta aplicacao com o modelo existente em
// outra aplicacao, no caso COMP011_MVC
Local oModel := FWLoadModel( "COMP011_MVC" )
Return oModel

//-----
Static Function ViewDef()
// Criamos o modelo de dados desta aplicacao com a interface existente em
// outra aplicacao, no caso COMP011_MVC
Local oView := FWLoadView( "COMP011_MVC" )
Return oView

```

## 19.2 Reusing and supplementing components

Now we will show how to reuse a MVC component in which we add new entities. If you can add new entities and not remove. If we remove any entities, we would be breaking the business rules created in the original model.

The ideal for this type of use is to create a basic model and increment it according to need.

We analyze the first data model (Model). In the example as from the existing the data model, we add a new entity.

The first step is to create the structure of the new entity, see chapter **0 5.1 Construction of a data structure (FWFormStruct)** for details.

```

// Cria a estrutura a ser acrescentada no Modelo de Dados
Local oStruZA6 := FWFormStruct( 1, 'ZA6', /*bAvalCampo*/, /*lViewUsado*/ )

```

We instantiate the existing model.

```
// Inicia o Model com um Model já existente
Local oModel := FWLoadModel( 'COMP011_MVC' )
```

In this example, we add a new form, see chapter. 0 5.3 Creation of a forms component in the data model (AddFields) for details.

Notice that in our new application we do not use the **MPFormModel**, as we are only adding an entity. **MPFormModel** was used in the original application.

```
// Adiciona a nova FORMFIELD
oModel:AddFields( 'ZA6MASTER', 'ZA0MASTER', oStruZA6 )
```

We perform the relationship of the new form, see chapter 0



## 6.5 Creation of relations among model entities (SetRelation).

```
// Faz relacionamento entre os componentes do model
oModel:SetRelation( 'ZA6MASTER', { { 'ZA6_FILIAL', 'xFilial( "ZA6" )' }, { 'ZA6_CODIGO',
'ZA0_CODIGO' } }, ZA6->( IndexKey( 1 ) ) )
```

We add the description of a new form.

```
// Adiciona a descricao do novo componente
oModel:GetModel( 'ZA6MASTER' ):SetDescription( 'Complemento dos Dados de Autor/Interprete'
```

At the end, we return to the new model.

```
Return oModel
```

With this, we create a model from another and add a new form component.

We will learn now how to reuse the interface (View), also adding a new component.

The first step is to create the structure of the new entity. See chapter 0 5.1 Construction of a data structure (FWFormStruct) .

```
// Cria a estrutura a ser acrescentada na View
Local oStruZA6 := FWFormStruct( 2, 'ZA6' )
```

We will instantiate the model used by the interface. Notice that we will not instantiate the original model by the new application model which already has a new component added in its data model.

```
// Cria um objeto de Modelo de Dados baseado no ModelDef do fonte informado
Local oModel := FWLoadModel( 'COMP015_MVC' )
```

We instantiate the original interface

```
// Inicia a View com uma View ja existente
Local oView := FWLoadView( 'COMP011_MVC' )
```

We added a new view component and associate it to the one created in the model, see chapter 0 5.8 Creation of a forms component on the interface (AddFields) for details.

```
// Adiciona no nosso View um controle do tipo FormFields(antiga enchoice)
oView:AddField( 'VIEW_ZA6', oStruZA6, 'ZA6MASTER' )
```

We have to create a **box** for the new component. You must always create a vertical **box** within a horizontal and vice-versa as **COMP011\_MVC**. The **box** already exists horizontally so first you must create a vertical. For details, see chapter **0 6.13 Display of data on interface (CreateHorizontalBox / CreateVerticalBox)**.

```
// 'TELANOVA' é o box existente na interface original
oView:CreateVerticalBox( 'TELANOVA' , 100, 'TELA' )
```

```
// Novos Boxes
```

```
oView:CreateHorizontalBox( 'SUPERIOR' , 50, 'TELANOVA' )  
oView:CreateHorizontalBox( 'INFERIOR' , 50, 'TELANOVA' )
```

Relating the component with the display **box**, see chapter. **0 5.10 Relating the interface component (SetOwnerView)**.

```
oView:SetOwnerView( 'VIEW_ZA0', 'SUPERIOR' )  
oView:SetOwnerView( 'VIEW_ZA6', 'INFERIOR' )
```

At the end, we return the interface's new object.

```
Return oView
```

With this, we create an interface from another and add a new component.

An example of application for this concept would be the internationalization, in which we could have a basic model and increment it according to the localization.

To better understand the internationalization, see Appendix A.

Below, we have the complete example of the application that reuses components.

## 19.3 Complete example of an application that reuses model and interface components

```
#INCLUDE 'PROTHEUS.CH'
#INCLUDE 'FWMVCDEF.CH'

//-----
User Function COMP015_MVC()
Local oBrowse

oBrowse := FWMBrowse():New()
oBrowse:SetAlias('ZA0')
oBrowse:SetDescription( 'Cadastro de Autor/Interprete' )
oBrowse:AddLegend( "ZA0_TIPO=='1'", "YELLOW", "Autor"          )
oBrowse:AddLegend( "ZA0_TIPO=='2'", "BLUE"   , "Interprete"      )
oBrowse:Activate()

Return NIL

//-----
Static Function MenuDef()
Local aRotina := {}
ADD OPTION aRotina TITLE 'Visualizar' ACTION 'VIEWDEF.COMP015_MVC' OPERATION 2 ACCESS 0
ADD OPTION aRotina TITLE 'Incluir'     ACTION 'VIEWDEF.COMP015_MVC' OPERATION 3 ACCESS 0
ADD OPTION aRotina TITLE 'Alterar'     ACTION 'VIEWDEF.COMP015_MVC' OPERATION 4 ACCESS 0
ADD OPTION aRotina TITLE 'Excluir'     ACTION 'VIEWDEF.COMP015_MVC' OPERATION 5 ACCESS 0
ADD OPTION aRotina TITLE 'Imprimir'    ACTION 'VIEWDEF.COMP015_MVC' OPERATION 8 ACCESS 0
ADD OPTION aRotina TITLE 'Copiar'      ACTION 'VIEWDEF.COMP015_MVC' OPERATION 9 ACCESS 0
Return aRotina

//-----
Static Function ModelDef()
// Cria a estrutura a ser acrescentada no Modelo de Dados
Local oStruZA6 := FWFormStruct( 1, 'ZA6', /*bAvalCampo*/,/*lViewUsado*/ )

// Inicia o Model com um Model ja existente
Local oModel := FWLoadModel( 'COMP011_MVC' )

// Adiciona a nova FORMFIELD
oModel:AddFields( 'ZA6MASTER', 'ZA0MASTER', oStruZA6 )
```

```

// Faz relacionamento entre os componentes do model
oModel:SetRelation( 'ZA6MASTER', { { 'ZA6_FILIAL', 'xFilial( "ZA6" )' }, { 'ZA6_CODIGO',
'ZA0_CODIGO' } }, ZA6->( IndexKey( 1 ) ) )

// Adiciona a descricao do novo componente
oModel:GetModel( 'ZA6MASTER' ):SetDescription( 'Complemento dos Dados de Autor/Interprete' )

Return oModel

//-----
Static Function ViewDef()
// Cria um objeto de Modelo de Dados baseado no ModelDef do fonte informado
Local oModel    := FWLoadModel( 'COMP015_MVC' )
// Cria a estrutura a ser acrescentada na View
Local oStruZA6 := FWFormStruct( 2, 'ZA6' )
// Inicia a View com uma View ja existente
Local oView     := FWLoadView( 'COMP011_MVC' )

// Altera o Modelo de dados quer será utilizado
oView:SetModel( oModel )
// Adiciona no nosso View um controle do tipo FormFields(antiga enchoice)
oView:AddField( 'VIEW_ZA6', oStruZA6, 'ZA6MASTER' )

// É preciso criar sempre um box vertical dentro de um horizontal e vice-versa
// como na COMP011_MVC o box é horizontal, cria-se um vertical primeiro
// Box existente na interface original
oView:CreateVerticallBox( 'TELANOVA' , 100, 'TELA'      )

// Novos Boxes
oView:CreateHorizontalBox( 'SUPERIOR' , 50, 'TELANOVA' )
oView:CreateHorizontalBox( 'INFERIOR' , 50, 'TELANOVA' )

// Relaciona o identificador (ID) da View com o "box" para exibicao
oView:SetOwnerView( 'VIEW_ZA0', 'SUPERIOR' )
oView:SetOwnerView( 'VIEW_ZA6', 'INFERIOR' )

Return oView

```

## Appendix A

The MVC Framework of Microsiga Protheus and the internationalization.

Internationalization (I18N) and localization (L10N) are development processes and/or software adaptation for a country's language and/or culture. The internationalization of a software does not provide a new System, it only adapts the System messages to the local language and culture. The localization adds news elements of the System country, such as processes, legal aspects, among others.

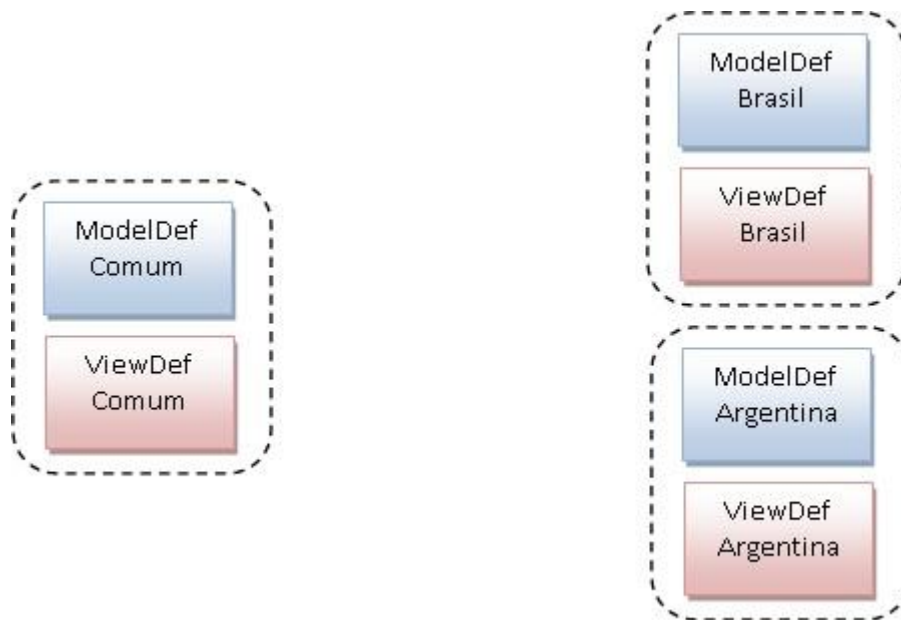
The *MVC* Framework helps the System localization, componentizing the software for the common part of all countries is disaggregated from the non-common part, adding the interface and business rule.

For example, take the o **Invoice** form as base. This form has the following elements as common feature in all countries: **Origin, Destination, List of products, Transportation and Invoices.**

In certain countries such as Brazil, you must register legal elements, **such as tax, bookkeeping, classification codes**, among others. The alternative is to duplicate the code or edit the code inserting the lines of codes of the elements localized. Although this alternative works very well in the beginning, over time it proves to be impractical due to the volume of different implementations for each country, causing great inconvenience and high costs to sustain the System.

MVC Framework provides a simple and rational light to this problem. The inheritance of forms. You can construct a common form for the **Invoice** which does not have any localization element and use it through inheritance as a base for the localized forms.

In this model, the evolution of the localization and the common part of the form are ensured without an implementation affecting the another by reducing the product sustainability cost.



The MVC framework inheritance can occur in the Model and View or only in the View.

At this point, you must be asking yourself how this can be done. The answer is in the **FWLoadModel** and **FWLoadView** functions, as you can see below in the code below:

```
#INCLUDE "MATA103BRA.CH"

Static Function ModelDef()
Local oModel := FWLoadModel("MATA103")
oModel:AddField(...)
oModel:AddGrid(...)
Return(oModel)

Static Function ViewDef()
Local oView := FWLoadView("MATA103")
oView:AddField(...)
oView:AddGrid(...)
Return (oView)
```

There are innumerable advantages in this development model that should be emphasized besides the componentization, which is the isolation of the source code. The isolation allows two source codes to evolve separately, but by inheritance, the localized code always inherits the benefits of the common part, including the possibility that two people interact simultaneously without one damaging the other's work.

# Table of Contents

AddCalc .....	64
AddField .....	18, 24, 58, 127
AddFields .....	16, 21, 126
AddGrid .....	21, 24, 30, 31
AddGroup .....	48
AddIncrementField .....	42
AddLegend .....	11
AddLine .....	34
AddOtherObjects .....	51
AddRules .....	41
AddTrigger .....	63
AddUserButton .....	43
AVG .....	65
AXALTERA .....	68
AXDELETA .....	68
AXINCLI .....	68
AXVISUAL .....	68
Campos de Total .....	64
CommitData .....	85, 91
Contadores .....	64
COUNT .....	65
CreateFolder .....	46
CreateHorizontalBox .....	18, 25, 127
CreateVerticalBox .....	18, 25, 127
DeleteLine .....	35
DisableDetails .....	12
EnableTitleView .....	44
ForceQuitButton .....	76
Fórmula .....	65
FWBrwRelation .....	77
FWBuildFeature .....	61
FWCalcStruct .....	67
FWExecView .....	68
FWFormCommit .....	40
FWFormStruct .....	14, 15, 16, 55, 126, 127
FWLoadMenuDef .....	70
FWLoadModel .....	17, 19, 24, 69, 124
FWLoadView .....	70, 124
FWMarkBrowse .....	71
FWMemoVirtual .....	62
FWModelActive .....	69
FWMVCMenu .....	9, 70
FWMVCRotAuto .....	92
FWRestRows .....	36
FWSaveRows .....	36
FwStruTrigger .....	63
FWViewActive .....	69
FwWsModel .....	101
Gatilhos .....	63
GetErrorMessage .....	85, 91
GetModel .....	29
GetOperation .....	39
GetSchema .....	105
GetValue .....	37
GetXMLData .....	104
GoLine .....	33

Help .....	28
Internacionalização .....	131
IsDeleted .....	33
IsInserted .....	33
IsMark .....	72
IsOptional .....	36
IsUpdated .....	33
Length .....	32
LinhaOk .....	30
LoadValue .....	38
Mark .....	72
MarkBrowse .....	71
Master-Detail .....	20
Mensagens .....	28
MenuDef .....	7, 8, 13
MODEL_OPERATION_DELETE .....	40
MODEL_OPERATION_INSERT .....	40
MODEL_OPERATION_UPDATE .....	40
ModelDef .....	7, 13, 15, 16, 20, 23, 27
Modelo1 .....	20, 111
Modelo2 .....	111
Modelo3 .....	27, 111
MPFormModel .....	15, 29
MSExecAuto .....	92
New Model .....	111
PARAMIXB .....	94
Pastas .....	46
Pontos De Entrada .....	16, 94
PutXMLData .....	105
RemoveField .....	56
Rotina Automática .....	82
SetDescription .....	16, 22
SetFieldAction .....	51
SetFilterDefault .....	12
SetNoDeleteLine .....	36
SetNoFolder .....	64
SetNoGroups .....	64
SetNoInsertLine .....	36
SetNoUpdateLine .....	36
SetOnlyQuery .....	39
SetOnlyView .....	39
SetOperation .....	84, 89
SetOptional .....	36
SetOwnerView .....	18, 26, 128
SetPrimaryKey .....	22
SetProfileID .....	76
SetProperty .....	56
SetRelation .....	22, 127
SetSemaphore .....	72
SetUniqueLine .....	30
SetValue .....	38
SetViewAction .....	50
SetViewProperty .....	45
SetVldActive .....	32
STRUCT_FEATURE_INIPAD .....	62
STRUCT_FEATURE_PICTVAR .....	62

STRUCT_FEATURE_VALID .....	62
STRUCT_FEATURE_WHEN .....	62
SUM .....	65
TudoOk .....	29
UnDeleteLine .....	35
Validações .....	29

ViewDef .....	7, 8, 17, 19, 24, 26, 27
VldData .....	85, 91
VldXMLData .....	104
WebServices .....	101
WsFwWsModel .....	101