Informativo

AdvPl utilizando MVC

A arquitetura *Model-View-Controller* ou MVC, como é mais conhecida, é um padrão de arquitetura de software que visa separar a lógica de negócio da lógica de apresentação (a interface), permitindo o desenvolvimento, teste e manutenção isolada de ambos.

Aqueles que já desenvolveram uma aplicação em AdvPL vão perceber, que justamente a diferença mais importante entre a forma de construir uma aplicação em MVC e a forma tradicional, é essa separação e que vai permitir o uso da regra de negócio em aplicações que tenham ou não interfaces, como *Web Services* e aplicação automática, bem como seu reuso em outras aplicações.

Índice

1.Arquitetura MVC

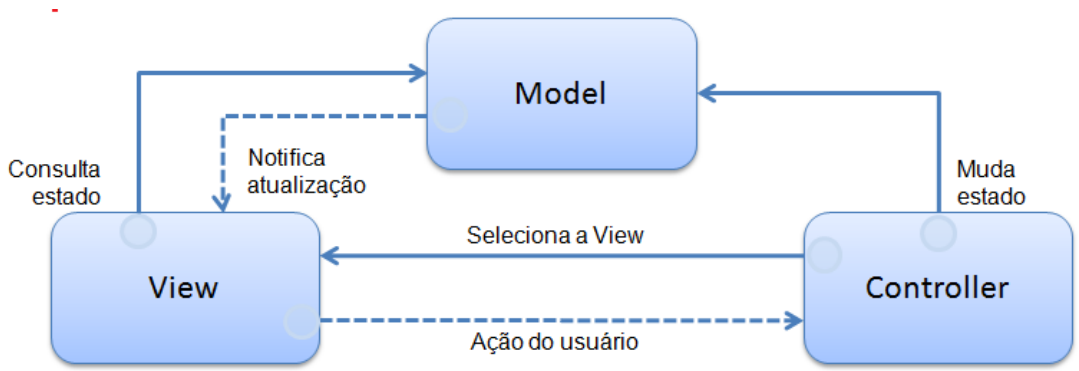
Primeiramente vamos entender o que é a arquitetura *MVC*.

A arquitetura ***Model-View-Controller*** ou ***MVC***, como é mais conhecida, é um padrão de arquitetura de software que visa separar a lógica de negócio da lógica de apresentação (a interface), permitindo o desenvolvimento, teste e manutenção isolados de ambos.

Aqueles que já desenvolveram uma aplicação em *AdvPL* vão perceber, que justamente a diferença mais importante entre a forma de construir uma aplicação em *MVC* e a forma tradicional é essa separação.

E é ela que vai permitir o uso da regra de negócio em aplicações que tenham ou não interfaces, como Web Services e aplicação automática, bem como seu reuso em outras aplicações.

A arquitetura *MVC* possui três componentes básicos:



***Model* ou modelo de dados:** representa as informações do domínio do aplicativo e fornece funções para operar os dados, isto é, ele contém as funcionalidades do aplicativo. Nele definimos as regras de negócio: tabelas, campos, estruturas, relacionamentos etc.. O modelo de dados *(Model)* também é responsável por notificar a *interface* (*View)* quando os dados forem alterados.

View* ou *interface: responsável por renderizar o modelo de dados *(Model)* e possibilitar a interação do usuário, ou seja, é o responsável por exibir os dados.

Controller: responde às ações dos usuários, possibilita mudanças no Modelo de dados *(Model)* e seleciona a *View* correspondente.

Para facilitar e agilizar o desenvolvimento, na implementação do *MVC* feita no *AdvPL*, o desenvolvedor trabalhará com as definições de Modelo de dados *(Model)* e *View*, a parte responsável pelo ***Controller*** já esta intrínseca.

Frisando bem, a grande mudança, o grande paradigma a ser quebrado na forma de pensar e se desenvolver uma aplicação em AdvPL utilizando MVC é a separação da regra de negócio da interface. Para que isso fosse possível foram desenvolvidas várias novas classes e métodos no AdvPL.

2.Principais funções da aplicação em *AdvPL* utilizando o *MVC*

Apresentamos agora o modelo de construção de uma aplicação em *AdvPL* utilizando o *MVC*.

Os desenvolvedores em suas aplicações serão responsáveis por definir as seguintes funções:

ModelDef: Contem a construção e a definição do *Model*, lembrando que o Modelo de dados *(Model)* contém as regras de negócio;

ViewDef: Contem a construção e definição da View, ou seja, será a construção da *interface*;

MenuDef: Contem a definição das operações disponíveis para o modelo de dados *(Model).*

Cada fonte em *MVC* (PRW) só pode conter uma de cada dessas funções. Só pode ter uma ***ModelDef***, uma ***ViewDef*** e uma ***MenuDef***.

Ao se fazer uma aplicação em *AdvPL* utilizando *MVC*, automaticamente ao final, esta aplicação já terá disponível:

- **Pontos de Entradas** já disponíveis;
- Um ***Web Service*** para sua utilização;
- **Importação ou exportação** mensagens XML.

Poderá ser utilizada, similarmente ao que é a rotina automática das aplicações sem *MVC.*

Um ponto importante na aplicação em *MVC* é que ela não se baseia necessariamente em metadados (dicionários). Como veremos mais a frente, ela se baseia em estruturas e essas por sua vez é que podem vir do metadados (dicionários) ou serem construídas manualmente.

2.1 O que é a função ModelDef?

A função ModelDef define a regra de negócios propriamente dita onde são definidas:

- **Todas as entidades** (tabelas) que farão parte do modelo de dados *(Model)*;
- **Regras de dependência** entre as entidades;
- **Validações** (de campos e aplicação);
- **Persistência** dos dados (gravação).

Para uma ***ModelDef*** não é preciso necessariamente haver uma *interface*. Como a regra de negócios é totalmente separada da *interface* no *MVC*, podemos utilizar a ***ModelDef*** em qualquer outra aplicação, ou até utilizarmos uma determinada ***ModelDef*** como base para outra mais complexa.

As entidades da ***ModelDef*** não se baseiam necessariamente em metadados (dicionários). Como veremos mais a frente, ela se baseia em estruturas e essas por sua vez é que podem vir do metadados ou serem construídas manualmente.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

A ***ModelDef*** deve ser uma ***Static Function*** dentro da aplicação.

2.2 O que é a função ViewDef?

A função ViewDef define como o será a *interface* e portanto como o usuário interage com o modelo de dados *(Model) r*ecebendo os dados informados pelo usuário, fornecendo ao modelo de dados (definido na ***ModelDef)*** e apresentando o resultado.

A *interface* pode ser baseada totalmente ou parcialmente em um metadado (dicionário), permitindo:

- **Reaproveitamento** do código da *interface*, pois uma *interface* básica pode ser acrescida de novos componentes;
- **Simplicidade** no desenvolvimento de *interfaces* complexas. Um exemplo disso são aquelas aplicações onde uma ***GRID*** depende de outra. No *MVC* a construção de aplicações que tem ***GRIDs*** dependentes é extremamente fácil;
- **Agilidade** no desenvolvimento, a criação e a manutenção se tornam muito mais ágeis;
- **Mais de uma** *interface* por *Bussiness Object*. Poderemos ter *interfaces* diferentes para cada variação de um segmento de mercado, como o varejo.

A ***ViewDef*** deve ser uma ***Static Function*** dentro da aplicação.

2.3 O que é a função MenuDef?

Uma função MenuDef define as operações quer serão realizadas pela aplicação, tais como **inclusão, alteração, exclusão**, etc..

Deve retornar um *array* em um formato específico com as seguintes informações:

1. Título;

2. Nome da aplicação associada;

3. Reservado;

4. Tipo de Transação a ser efetuada.

E que podem ser:

- 1 para Pesquisar

- 2 para Visualizar

- 3 para Incluir

- 4 para Alterar

- 5 para Excluir

- 6 para Imprimir

- 7 para Copiar

5. Nível de acesso;

6. Habilita Menu Funcional;

Exemplo:

```
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
```

Note que o 2º parâmetro utiliza a chamada direta de uma aplicação, ela faz referência a uma *ViewDef* de um determinado fonte (PRW).

A estrutura deste 2º parâmetro tem o formato:

ViewDef.<nome do fonte>

Sempre referenciaremos a ***ViewDef*** de um fonte, pois ela é a função responsável pela a *interface* da aplicação.

Para facilitar o desenvolvimento, no *MVC* a ***MenuDef*** escreva-a da seguinte forma:

```
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
```

O resultado final é o mesmo, o que difere é apenas a forma de construção, mas **é recomendado a 2ª forma** que utiliza o formato de comandos e não posições de um vetor, pois uma eventual manutenção se tornará mais fácil.

A ***MenuDef*** deve ser uma ***Static Function*** dentro da aplicação.

Utilizando-se a função ***FWMVCMenu***, obtêm-se um menu padrão com as opções: Visualizar, Incluir, Alterar, Excluir, Imprimir e Copiar. Deve ser passado como parâmetro no nome do fonte.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Por exemplo:

```
Static Function MenuDef()
Return FWMVCMenu( "COMP021_MVC" )
```

Isso criaria um ***Menudef*** exatamente como o exemplo anterior. Para mais detalhes veja o capítulo **12.7 Obter um menu padrão (*FWMVCMenu*).**

2.4 Novo comportamento na interface

Nas aplicações desenvolvidas em *AdvPL* tradicional, após a conclusão de uma operação de alteração fecha-se a *interface* e retorna ao *Browse*.

Nas aplicações em *MVC,* após as operações de inclusão e alteração, a *interface* permanece ativa e no rodapé exibe-se a mensagem de que a operação foi bem sucedida.

3.Aplicações com Browses (*FWMBrowse*)

Para a construção de uma aplicação que possui um ***Browse,*** o *MVC* utiliza a classe ***FWMBrowse***.

Esta classe exibe um objeto ***Browse*** que é construído a partir de metadados (dicionários).

Esta classe **não foi desenvolvida exclusivamente** para o *MVC*, aplicações que não são em *MVC* também podem utilizá-la. No *MVC* a utilizaremos.

Suas **características** são:

- Substituir componentes de *Browse;*
- Reduzir o tempo de manutenção, em caso de adição de um novo requisito;
- Ser independente do ambiente Microsiga Protheus.

E apresenta como **principais melhorias**:

- Padronização de legenda de cores;
- Melhor usabilidade no tratamento de filtros;
- Padrão de cores, fontes e legenda definidas pelo usuário – Deficiente visual;
- Redução do número de operações no SGBD (no mínimo 3 vezes mais rápido);
- Novo padrão visual.

3.1 Construção de um Browse

Falaremos aqui de principais funções e características para uso em aplicações com *MVC*.

3.2 Construção básica de um Browse

Iniciamos a construção básica de um *Browse.*

Primeiramente crie um objeto *Browse* da seguinte forma:

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

Definimos a tabela que será exibida na *Browse* utilizando o método ***SetAlias***. As colunas, ordens, etc..

A exibição é obtidos pelo metadados (dicionários).

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

Definimos o título que será exibido como método ***SetDescription.***

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

E ao final ativamos a classe.

```
oBrowse:Activate()
```

Com esta estrutura básica construímos uma aplicação com *Browse*.

O *Browse* apresentado automaticamente já terá:

- Pesquisa de registro;
- Filtro configurável;
- Configuração de colunas e aparência;
- Impressão.

3.3 Legendas de um Browse (AddLegend)

Para o uso de legendas no *Browse* utilizamos o método ***AddLegend***, que possui a seguinte sintaxe:

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

Exemplo:

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

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

cRegra é a expressão em *AdvPL* para definir a legenda.

cCor é o parâmetro que define a cor de cada item da legenda.

São possíveis os seguintes valores:

GREEN	Para a cor Verde
RED	Para a cor Vermelha
YELLOW	Para a cor Amarela
ORANGE	Para a cor Laranja
BLUE	Para a cor Azul
GRAY	Para a cor Cinza
BROWN	Para a cor Marrom
BLACK	Para a cor Preta

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

PINK Para a cor Rosa

WHITE Para a cor Branca

cDescrição é a que será exibida para cada item da legenda

Observação:

- Cada uma das legendas se tornará automaticamente uma opção de filtro.
- Cuidado ao montar as regras da legenda. Se houverem regras conflitantes será exibida a legenda correspondente à 1ª regra que for satisfeita.

3.4 Filtros de um Browse (SetFilterDefault)

Se quisermos definir um filtro para o *Browse* utilizamos o método ***SetFilterDefault***, que possui a seguinte sintaxe:

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

Exemplo:

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

ou

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

A expressão de filtro é em AdvPL.

O filtro definido na aplicação não anula a possibilidade do usuário fazer seus próprios filtros. Os filtros feitos pelo usuário serão aplicados em conjunto com o definido na aplicação (condição de **AND**).

Exemplo:

Se na aplicação foi definido que só serão exibidos clientes que são pessoas jurídicas, se usuário fizer um filtro para exibir apenas clientes do estado de São Paulo, serão exibidos os clientes pessoa jurídica do estado de São Paulo. Foi executado o filtro do usuário e ainda respeitado o filtro original da aplicação.

Observação: O filtro da aplicação não poderá ser desabilitado pelo usuário.

3.5 Desabilitação de detalhes do Browse (DisableDetails)

Automaticamente para o ***Browse*** são exibidos, em detalhes, os dados da linha posicionada. Para desabilitar esta característica utilizamos o método ***DisableDetails.***

Exemplo:

```
oBrowse:DisableDetails()
```

3.6 Campos virtuais no Browse

Normalmente, para se exibir campos virtuais nos *Browses*, fazemos uso da função ***Posicione.***

No novo Browse esta prática se torna ainda mais importante, pois, quando ele encontra a função ***Posicione*** na definição de um campo virtual e a base de dados é um ***SGBD*** (usa o ***TOTVSDbAccess***), o Browse acrescenta um ***INNER JOIN*** na *query* que será enviada ao ***SGBD***, melhorando assim o desempenho para a extração dos dados.

Portanto, sempre utilize a função ***Posicione*** para exibir campos virtuais.

3.7 Exemplo completo de 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'" )

// Titulo 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.Construção de aplicação *AdvPL* utilizando *MVC*

Iniciamos agora a construção da parte em *MVC* da aplicação, que são as funções de ***ModeDef***, que contem as regras de negócio e a ***ViewDef*** que contem a *interface*.

Um ponto **importante** que deve ser observado é que, assim como a ***MenuDef***, só **pode haver uma função *ModelDef*** e uma **função *ViewDef*** em um fonte**.**

Se para uma determinada situação for preciso trabalhar em mais de um modelo de dados *(Model)*, a aplicação deve ser quebrada em vários fontes (PRW) cada um com apenas uma ***ModelDef*** e uma ***ViewDef***.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

5.Construção de aplicação *MVC* com uma entidade

Mostramos como criar uma aplicação em *MVC* com apenas uma entidade envolvida.

5.1 Construção de uma estrutura de dados (FWFormStruct)

A primeira coisa que precisamos fazer é criar a estrutura utilizada no modelo de dados *(Model)*.

As estruturas são objetos que contêm as definições dos dados necessárias para uso da ***ModelDef*** ou para a ***ViewDef.***

Esses objetos contêm:

- Estrutura dos Campos;
- Índices;
- Gatilhos;
- Regras de preenchimento (veremos à frente);
- Etc.

Como dito anteriormente o *MVC* não trabalha vinculado aos metadados (dicionários) do Microsiga Protheus, ele trabalha vinculado a estruturas. Essas estruturas, por sua vez, é que podem ser construídas a partir dos metadados.

Com a função ***FWFormStruct*** a estrutura será criada a partir do metadado.

Sua sintaxe é :

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

Onde:

nTipo Tipo da construção da estrutura: 1 para Modelo de dados *(Model)* e 2 para *interface (View);*

cAlias Alias da tabela no metadado;

Exemplo:

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

No exemplo, o objeto ***oStruZA0*** será uma estrutura para uso em um modelo de dados *(Model)*. O primeiro parâmetro (1) indica que a estrutura é para uso no modelo e o segundo parâmetro indica qual a tabela dos metadados será usada para a criação da estrutura (*ZA0*).

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

No exemplo dado, o objeto ***oStruZA0*** será uma estrutura para uso em uma *interface (View)*. O primeiro parâmetro (2) indica que a estrutura é para uso em uma *interface* e o segundo parâmetro indica qual a tabela dos metadados será usada para a criação da estrutura (*ZA0*).

Mais adiante veremos como criar estruturas manualmente e como selecionar os campos que

farão parte das estruturas e outros tratamentos específicos da estrutura.

> **Importante**: Para modelo de dados *(Model)*, a função ***FWFormStruct***, traz para a estrutura todos os campos que compõem a tabela independentemente do nível, uso ou módulo. Considera também os campos virtuais.
>
> Para a *interface (View)* a função ***FWFormStruct***, traz para a estrutura os campos conforme o nível, uso ou módulo.

5.2 Construção da função ModelDef

Como foi dito anteriormente, nesta função são definidas as regras de negócio ou modelo de dados *(Model)*.

Ela contêm as definições de:

- Entidades envolvidas;
- Validações;
- Relacionamentos;
- Persistência de dados (gravação);
- Etc.

Iniciamos a função ***ModelDef***:

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

Construindo o Model

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

MPFormModel é a classe utilizada para a construção de um objeto de modelo de dados *(Model).*

Devemos dar um identificador *(ID)* para o modelo como um todo e também um para cada componente.

Essa é uma característica do MVC, todo componente do modelo ou da *interface* devem ter um ID, como formulários, ***GRIDs***, ***boxes***, etc.

COMP011M é o identificador *(ID)* dado ao Model, é importante ressaltar com relação ao identificador *(ID)* do Model:

- Se a aplicação é uma ***Function,*** o identificador *(ID)* do modelo de dados *(Model)* pode ter o mesmo nome da função principal e esta prática é recomendada para facilitar a codificação. Por exemplo, se estamos escrevendo a função XPTO, o identificador *(ID)* do modelo de dados *(Model)* poderá ser XPTO.

- Se a aplicação é uma ***User Function*** o identificador *(ID)* do modelo de dados *(Model)* **NÃO** pode ter o mesmo nome da função principal, isso por causa dos pontos de

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

entrada que já são criados automaticamente quando desenvolvemos uma aplicação em MVC. Isso será mais detalhado à frente (ver capítulo **16.Pontos de entrada no MVC**).

5.3 Criação de um componente de formulários no modelo de dados (AddFields)

O método ***AddFields*** adiciona um componente de formulário ao modelo.

A estrutura do modelo de dados *(Model)* deve iniciar, obrigatoriamente, com um componente de formulário.

Exemplo:

```
oModel:AddFields( 'ZA0MASTER', /*cOwner*/, oStruZA0 )
```

Devemos dar um identificador *(ID)* para cada componente do modelo.

ZA0MASTER é o identificador *(ID)* dado ao componente de formulário no modelo, ***oStruZA0*** é a estrutura que será usada no formulário e que foi construída anteriormente utilizando ***FWFormStruct***, note que o segundo parâmetro (*owner*) não foi informado, isso porque este é o 1º componente do modelo, é o **Pai** do modelo de dados *(Model)* e portanto não tem um componente superior ou ***owner***.

5.4 Descrição dos componentes do modelo de dados (SetDescription)

Sempre definindo uma descrição para os componentes do modelo.

Com o método ***SetDescription*** adicionamos a descrição ao modelo de dados *(Model)*, essa descrição será usada em vários lugares como em *Web Service*s por exemplo.

Adicionamos a descrição do **modelo de dados**:

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

Adicionamos a descrição dos **componentes do modelo de dados:**

```
oModel:GetModel( 'ZA0MASTER' ):SetDescription( 'Dados de Autor/Interprete' )
```

Para um modelo que só contêm um componente parece ser redundante darmos uma descrição para o modelo de dados *(Model)* como um todo e uma para o componente, mas quando estudarmos outros modelos onde haverá mais de um componente esta ação ficará mais clara.

5.5 Finalização de ModelDef

Ao final da função ***ModelDef***, deve ser retornado o objeto de modelo de dados *(Model)* gerado na função.

```
Return oModel
```

5.6 Exemplo completo da 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( 'ZA0MASTER', /*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( 'ZA0MASTER' ):SetDescription( 'Dados de Autor/Interprete' )

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

5.7 Construção da função ViewDef

A *interface (View)* é responsável por renderizar o modelo de dados *(Model)* e possibilitar a interação do usuário, ou seja, é o responsável por exibir os dados.

O ***ViewDef*** contêm a definição de toda a parte visual da aplicação.

Iniciamos a função:

```
Static Function ViewDef()
```

A *interface (View)* sempre trabalha baseada em um modelo de dados *(Model)*. Criaremos um objeto de modelo de dados baseado no ***ModelDef*** que desejamos.

Com a função ***FWLoadModel*** obtemos o modelo de dados *(Model)* que está definido em um fonte, no nosso caso é o próprio fonte, mas nada impediria que usássemos o modelo de qualquer outro fonte em *MVC*, com isso podemos reaproveitar um mesmo modelo de dados *(Model)* em mais de uma *interface (View)*.

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

COMP011_MVC é nome do fonte de onde queremos obter o modelo de dados *(Model).*

Iniciando a construção da *interface (View)*

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

FWFormView é a classe que deverá ser usada para a construção de um objeto de *interface (View)*.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Definimos qual o modelo de dados *(Model)* que será utilizado na *interface (View).*

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

5.8 Criação de um componente de formulários na interface (AddField)

Adicionamos no nosso *interface (View)* um controle do tipo formulário (antiga ***Enchoice***), para isso usamos o método ***AddField***

A *interface (View)* deve iniciar, obrigatoriamente, com um componente do tipo formulário.

```
oView:AddField( 'VIEW_ZA0', oStruZA0, 'ZA0MASTER' )
```

Devemos dar um identificador *(ID)* para cada componente do *interface (View)*.

VIEW_ZA0 é o identificador *(ID)* dado ao componente da *interface (View)*, ***oStruZA0*** é a estrutura que será usada e ***ZA0MASTER*** é identificador *(ID)* do componente do modelo de dados *(Model)* vinculado a este componente da *interface (View)*.

Cada componente da *interface (View)* deve ter um componente do modelo de dados *(Model)* relacionado, isso equivale a dizer que os dados do ***ZA0MASTER*** serão exibidos na *interface (View)* no componente ***VIEW_ZA0***

5.9 Exibição dos dados na interface (CreateHorizontalBox / CreateVerticalBox)

Sempre precisamos criar um ***contêiner***[1], um objeto, para receber algum elemento da *interface (View)*. Em *MVC* criaremos sempre ***box*** horizontal ou vertical para isso.

O método para criação de um ***box*** horizontal é:

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

Devemos dar um identificador *(ID)* para cada componente da *interface (View)*.

TELA é o identificador *(ID)* dado ao ***box*** e o número **100** representa o percentual da tela que será utilizado pelo Box.

No MVC não há referências a coordenadas absolutas de tela, os componentes visuais são sempre **All Client**, ou seja, ocuparão todo o ***contêiner*** onde for inserido

5.10 Relacionando o componente da interface (SetOwnerView)

Precisamos relacionar o componente da *interface (View)* com um ***box*** para exibição, para isso usamos o método ***SetOwnerView.***

[1] Determinada área definida pelo desenvolvedor para agrupar componentes visuais, por exemplo, Panel, Dialog, Window, etc

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

Desta forma o componente ***VIEW_ZA0*** será exibido na tela utilizando o box ***TELA.***

5.11 Finalização da ViewDef

Ao final da função ***ViewDef***, deve ser retornado o objeto de *interface (View)* gerado

```
Return oView
```

5.12 Exemplo completo da ViewDef

```
Static Function ViewDef()
// Cria um objeto de Modelo de dados baseado no ModelDef() do fonte informado
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, 'ZA0MASTER' )

// 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 Finalização da criação da aplicação com uma entidade

Desta forma criamos uma aplicação de *AdvPL* utilizando MVC onde há apenas uma entidade envolvida.

- Construímos a ***ModelDef;***

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

- Construímos a ***ViewDef.***

Esta aplicação seria o equivalente às aplicações de tipo ***Modelo1*** que normalmente são feitas.

Veremos a seguir a construção de aplicações utilizando duas ou mais entidades.

6.Construção de uma aplicação *MVC* com duas ou mais entidades

Vimos até agora a construção de uma aplicação onde era utilizada apenas uma entidade. Veremos a construção onde duas ou mais entidades irão existir.

A construção da aplicação seguirá os mesmos passos que vimos até agora: Construção da ***ModelDef*** e da ***ViewDef***. A diferença básica é que agora cada uma delas possuirá mais de um componente e eles se relacionarão.

6.1 Construção de estruturas para uma aplicação MVC com duas ou mais entidades

Como descrevemos, a primeira coisa que precisamos fazer é criar a estrutura utilizada no modelo de dados (*Model*). Temos que criar uma estrutura para cada entidade que participará do modelo. Se forem 2 entidades, 2 estruturas, se forem 3 entidades, 3 estruturas e assim por diante.

Mostraremos uma aplicação onde temos 2 entidades em uma relação de dependência de ***Master-Detail*** (**Pai-Filho**), como por exemplo um Pedido de Venda, onde temos o cabeçalho do pedido seria o ***Master*** (**Pai**) e os itens seriam o ***Detail*** (**Filho**)

A construção das estruturas seria:

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

No exemplo anterior o objeto ***oStruZA1*** será uma estrutura para ser utilizada em um Modelo de dados *(Model)* para a entidade ***Master*** (**Pai**) e ***oStruZA2*** para a entidade ***Detail*** (**Filho**).

O primeiro parâmetro (1) indica que a estrutura é para ser utilizada em um modelo de dados *(Model)* e segundo indica qual a tabela será usada para a criação da estrutura.

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

No exemplo acima o objeto ***oStruZA1*** será uma estrutura para ser utilizada em uma *interface (View)* para a entidade ***Master*** (**Pai)** e ***oStruZA2*** para a entidade ***Detail*** (**Filho**). O primeiro parâmetro (2) indica que a estrutura é para ser utilizada em uma *interface (View)* e o segundo indica qual tabela será usada para a criação da estrutura.

6.2 Construção de uma função ModelDef

Iniciamos a função ***ModelDef***.

```
Static Function ModelDef()
```

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

Observe que no código, houve a criação de 2 estruturas uma para cada entidade.

Começamos a construção do Model

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

Devemos dar um identificador *(ID)* para o Modelo de dados *(Model)* e para cada componente do Model.

COMP021M é o identificador *(ID)* dado ao Modelo de dados *(Model).*

6.3 Criação de um componente de formulários no modelo de dados (AddFields)

O método ***AddFields*** adiciona ao modelo um componente de formulário.

A estrutura do modelo deve iniciar, obrigatoriamente, com um componente de formulário.

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

Devemos dar um identificador *(ID)* para cada componente do Model.

ZA1MASTER é o identificador *(ID)* dado ao formulário no Model, ***oStruZA1*** é a estrutura que será usada no formulário e que foi construída anteriormente utilizando ***FWFormStruct;*** note que o segundo parâmetro (*Owner*) não foi informado, isso porque este é o 1º componente do Model, é o **Pai** do modelo de dados *(Model)* e, portanto não tem um componente superior ou ***owner***.

6.4 Criação de um componente de grid no Modelo de dados (AddGrid)

A relação de dependência entre as entidades é de ***Master-Detail***, ou seja, há 1 ocorrência do **Pai** para ***n*** ocorrências do **Filho** (1-*n*)

Quando uma entidade ocorrerá ***n*** vezes no modelo em relação à outra, devemos definir um componente de ***Grid*** para esta entidade.

O método ***AddGrid*** adiciona ao modelo um componente de *grid.*

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

Devemos dar um identificador *(ID)* para cada componente do Model.

ZA2DETAIL é o identificador *(ID)* dado ao componente no Model, ***oStruZA2*** é a estrutura que será usada no componente e que foi construída anteriormente utilizando ***FWFormStruct***, note que o segundo parâmetro (*Owner*) desta vez foi informado, isso porque esta entidade depende da 1ª (*Master*), portanto ***ZA1MASTER*** é o componente superior ou ***owner*** de ***ZA2DETAIL.***

6.5 Criação de relação entre as entidades do modelo (SetRelation)

Dentro do modelo devemos relacionar todas as entidades que participam dele. No nosso exemplo temos que relacionar a entidade ***Detail*** com a entidade ***Master.***

Uma regrinha bem simples para entender isso é: Toda entidade do modelo que possui um superior (*owner*) dever ter seu relacionamento para ele definido. Em outras palavras, é preciso dizer quais as chaves de relacionamento do filho para o pai.

O método utilizado para esta definição é o ***SetRelation.***

Exemplo:

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

O ***ZA2DETAIL*** é o identificador *(ID)* da entidade ***Detail***, o segundo parâmetro é um vetor bi-dimensional onde são definidos os relacionamentos entre cada campo da entidade filho para a entidade Pai. O terceiro parâmetro é a ordenação destes dados no componente.

Estamos dizendo no exemplo acima que o relacionamento da entidade Detail será por ***ZA2_FILIAL*** e ***ZA2_MUSICA***, o valor de ***ZA2_FILIAL*** será dado por ***xFilial()*** e o de ***ZA2_MUSICA*** virá de ***ZA1_MUSICA.***

Observação: O relacionamento sempre é definido do ***Detail*** (**Filho**) para o ***Master*** (**Pai**), tanto no identificador *(ID)* quanto na ordem do vetor bi-dimensional.

6.6 Definição da chave primária (SetPrimaryKey)

O modelo de dados precisa que sempre se informe qual a chave primária para a entidade principal do modelo de dados (Model).

Se a estrutura da entidade foi construída utilizando-se o **FWFormStruct**, a chave primária será aquela que foi definida no metadados (dicionários).

Se a estrutura foi construída manualmente ou se a entidade não possui definição de chave única no metadados, temos que definir qual será essa chave com o método S **SetPrimaryKey**.

Exemplo:

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

Onde o parâmetro passado é um vetor com os campos que compõem a chave primária. **Use este método somente se for preciso.**

Sempre defina a chave primária para o modelo. Se realmente não for possível criar uma chave primária para a entidade principal, informe-o no modelo de dados da seguinte forma:

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

6.7 Descrevendo os componentes do modelo de dados (SetDescription)

Defina sempre uma descrição para os componentes do modelo. Com o método ***SetDescription***

adicionamos a descrição do Modelo de Dados, essa descrição será usada em vários lugares como em *Web Services* por exemplo.

Adicionamos a descrição do modelo de dados.

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

Adicionamos a descrição dos componentes do modelo de dados.

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

Note que desta vez definimos uma descrição para modelo e uma para cada componente do modelo.

6.8 Finalização da ModelDef

Ao final da função ***ModelDef***, deve ser retornado o objeto de Modelo de dados *(Model)* gerado na função.

```
Return oModel
```

6.9 Exemplo completo da 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 Construção da função ViewDef

Iniciamos a função.

```
Static Function ViewDef()
```

A *interface (View)* sempre trabalhará baseada em um modelo de dados (*Model)*.

Criamos um objeto de Modelo de dados baseado no ***ModelDef*** que desejamos.

Com a função ***FWLoadModel*** obtemos o modelo de dados *(Model)* que está definido em um fonte, no nosso caso, é o próprio fonte, mas nada impede que usássemos o modelo de dados *(Model)* de qualquer outro fonte em *MVC*, com isso podemos reaproveitar um mesmo Modelo de dados *(Model)* em mais de uma *interface (View)*.

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

COMP021_MVCé nome do fonte de onde queremos obter o *model.*

Começamos a construção da *interface (View)*

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

FWFormView é a classe que deverá ser usada para a construção de um objeto de *interface (View)*.

Definimos qual o Modelo de dados *(Model)* que será utilizado na *interface (View).*

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

6.11 Criação de um componente de formulários na interface (AddField)

Adicionamos na nossa *interface (View)* um controle do tipo formulário (antiga ***Enchoice***), para isso usamos o método ***AddField.***

A *interface (View)* deve iniciar, obrigatoriamente, com um componente do tipo formulário.

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

Devemos dar um identificador *(ID)* para cada componente do *interface (View)*. ***VIEW_ZA1*** é o identificador *(ID)* dado ao componente da *interface (View)*, ***oStruZA1*** é a estrutura que será usada e ***ZA1MASTER*** é identificador *(ID)* do componente do Modelo de dados *(Model)* vinculado a este componente da *interface (View)*.

Cada componente da *interface (View)* deve ter um componente do Modelo de dados *(Model)* relacionado, isso equivale a dizer que os dados do ***ZA1MASTER*** serão exibidos na *interface (View)* no componente ***VIEW_ZA1.***

6.12 Criação de um componente de grid na interface (AddGrid)

Adicionamos no nosso *interface (View)* um controle do tipo *grid* (antiga ***GetDados***), para isso usamos o método ***AddGrid.***

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

Devemos dar um identificador *(ID)* para cada componente do *interface (View)*.

VIEW_ZA2 é o identificador *(ID)* dado ao componente da *interface (View)*, ***oStruZA2*** é a estrutura que será usada e ***ZA2DETAIL*** é identificador *(ID)* do componente do Modelo de dados *(Model)* vinculado a este componente da *interface (View)*.

Cada componente da *interface (View)* deve ter um componente do Modelo de dados *(Model)* relacionado, isso equivale a dizer que os dados do ***ZA2DETAIL*** serão exibidos na *interface (View)* no componente ***VIEW_ZA2.***

Observação: Note que aqui não falamos que entidade é superior a qual, isso porque esta função é do modelo de dados. A *interface (View)* só reflete os dados do modelo.

6.13 Exibição dos dados na interface (CreateHorizontalBox / CreateVerticalBox)

Sempre precisamos criar um contêiner, um objeto, para receber algum elemento da *interface (View)*.

Em *MVC* criaremos sempre ***box*** horizontal ou vertical para isso.

O método para criação de um box horizontal é:

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

Devemos dar um identificador *(ID)* para cada componente do *interface (View)*. ***SUPERIOR*** é o identificador *(ID)* dado ao ***box*** e número **15** representa o percentual da tela que será utilizado pelo ***box***.

Como teremos dois componentes precisamos definir mais um box para o segundo componente

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

INFERIOR é o identificador *(ID)* dado ao ***box*** e número **85** representa o percentual da tela que será utilizado por ele.

Observação: A soma dos percentuais dos boxes de mesmo nível dever ser sempre 100%.

6.14 Relacionando o componente da interface (SetOwnerView)

Precisamos relacionar o componente da *interface (View)* com um **box** para exibição, para isso usamos o método ***SetOwnerView.***

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

Desta forma o componente ***VIEW_ZA1*** será exibido na tela pelo box ***SUPERIOR*** e o componente ***VIEW_ZA2*** será exibido na tela pelo box ***INFERIOR.***

Obs.: Note que os dados da entidade **Pai** ocuparão 15% da tela e da entidade **Filho** 85%, pois:

Id do Model	Id da View	Id do Box
ZA1MASTER	VIEW_ZA1	SUPERIOR (15%)
ZA2DETAIL	VIEW_ZA2	INFERIOR (85%)

6.15 Finalização da ViewDef

Ao final da função ***ViewDef***, deve ser retornado o objeto de *interface (View)* gerado.

```
Return oView
```

6.16 Exemplo completo da 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 Finalização da criação da aplicação com duas ou mais entidades

Desta forma criamos uma aplicação de *AdvPL* utilizando *MVC* onde há 2 entidades envolvidas.

- Construímos a ***ModelDef;***
- Construímos a ***ViewDef.***

Esta aplicação seria o equivalente às aplicações de tipo ***Modelo3*** que normalmente são feitas.

Se a necessidade for a construção de uma aplicação com mais de 2 entidades o processo será o mesmo que o mostrado para 2. A diferença será somente a quantidade de cada componente ou objeto que serão criados.

Para o modelo de dados *(Model)* se a aplicação tem 3 entidades, serão precisos 3 estruturas, 3 componentes ***AddFields*** ou ***AddGrid*** e 2 relacionamentos. Se a aplicação tem 4 entidades, serão precisos 4 estruturas, 4 componentes ***AddFields*** ou ***AddGrid*** e 3 relacionamentos e assim por diante.

Para a *interface (View)* se a aplicação tem 3 entidades, serão precisos 3 estruturas, 3 componentes ***AddField*** ou ***AddGrid*** e 3 boxes. Se a aplicação tem 4 entidades, serão precisos 4 estruturas, 4 componentes ***AddField*** ou ***AddGrid*** e 4 boxes e assim por diante.

O modelo de dados e a *interface* crescem na medida em que cresce a quantidade de entidades relacionadas. Porém a forma básica de construção é sempre a mesma.

7.Tratamentos para o modelo de dados e para *interface*

Agora que já sabemos como construir uma para aplicação em *MVC* utilizando ***n*** entidades, o que demonstraremos neste capítulo são os tratamentos específicos para algumas necessidades na construção de uma aplicação para a regra de negócio e para *interface*, pois em termos de hierarquia a idéia é sempre a mesma.

Exemplo:

- Validações;
- Permissões;
- Movimentação em linhas;
- Obter e atribuir valores;
- Persistência dos dados;
- Criar botões;
- Criar folders; etc.

8.Tratamentos para o modelo de dados

Veremos alguns tratamentos que podem ser feitos no modelo de dados *(Model)* conforme a necessidade:

- Validações;
- Comportamentos;
- Manipulação da *Grid*.
- Obter e atribuir valores ao modelo de dados *(Model)*;
- Gravação dos dados manualmente;
- Regras de preenchimento.

8.1 Mensagens exibidas na interface

As mensagens são usadas principalmente durante as validações feitas no modelo de dados.

Vamos analisar: Um ponto básico do *MVC* é a separação da regra de negócio da *interface*.

A validação é um processo executado dentro da regra de negócio e uma eventual mensagem de erro que será exibida ao usuário, é um processo que deve ser executado na *interface*, ou seja, não pode ser executado na regra de negócios.

Para trabalhar essa situação foi feito um tratamento para a função ***Help***.

A função ***Help*** poderá ser utilizada nas funções dentro do modelo de dados (*Model*), porém o *MVC* ira guardar essa mensagem e ela só será exibida quando o controle voltar para a *interface*.

Por exemplo, uma determinada função conterá:

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

Supondo que a mensagem de erro foi acionada porque o preço unitário é 0 (zero), neste momento não será exibido nada ao usuário, isso pode ser observado ao *debugar* o fonte. Você verá que ao passar pela função ***Help*** nada acontece, porém, quando o controle interno volta para a *interface*, a mensagem é exibida.

Esse tratamento foi feito apenas para a função ***Help***, funções como ***MsgStop***, ***MsgInfo***, ***MsgYesNo***, ***Alert***, ***MostraErro,*** etc. **não poderão** ser utilizadas.

8.2 Obtenção de componente do modelo de dados (GetModel)

Durante o desenvolvimento várias vezes teremos que manipular o modelo de dados *(Model)*, para facilitar essa manipulação podemos ao invés de trabalhar como o modelo todo, trabalhar com uma parte específica (um componente) de cada vez.

Para isso utilizamos o método ***GetModel.***

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

oModelZA2 é o objeto que contém um componente do modelo de dados *(Model)* e ***ZA2DETAIL*** é o identificador *(ID)* do componente que queremos.

Se tivermos uma parte do modelo de dados *(Model)* e quisermos pegar o modelo completo também podemos usar o ***GetModel.***

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

oModel é o objeto que contém o modelo de dados *(Model)* completo.

8.3 Validações

Dentro do modelo de dados existentes vários pontos onde podem ser inseridas as validações necessárias à regra de negócio. O modelo de dados *(Model)* como um todo tem seus pontos e cada componente do modelo também.

8.3.1 Pós-validação do modelo

É a validação realizada após o preenchimento do modelo de dados *(Model)* e sua confirmação. Seria o equivalente ao antigo processo de ***TudoOk.***

O modelo de dados *(Model)* já faz a validação se os campos obrigatórios de todos os componentes do modelo foram preenchidos, essa validação é executada após isso.

Definimos a pos-validação do modelo de dados *(Model)* como um bloco de código no 3º parâmetro da classe de construção do modelo ***MPFormModel.***

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


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

O bloco de código recebe como parâmetro um objeto que é o modelo e que pode ser passado à função que fará a validação.

```
Static Function COMP011POS( oModel )


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

A função chamada pelo bloco de código deve retornar um valor lógico, onde se .T. (verdadeiro) a operação é realizada e .F. (falso) não é realizada.

8.3.2 Pós-validação de linha

Em um modelo de dados (*Model*) onde existam componentes de *grid,* pode ser definida uma validação que será executada na troca das linhas do *grid*. Seria o equivalente ao antigo processo de ***LinhaOk.***

Definimos a pos-validação de linha como um bloco de código no 5º parâmetros do método ***AddGrid.***

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

O bloco de código recebe como parâmetro um objeto que é a parte do modelo correspondente apenas ao *grid* e que pode ser passado para a função que fará a validação.

A função chamada pelo bloco de código deve retornar um valor lógico, onde se .T. (verdadeiro) a troca de linha é realizada e .F. (falso) não é realizada.

8.3.3 Validação de linha duplicada (SetUniqueLine)

Em um modelo de dados onde existam componentes de *grid* podem ser definidos quais os campos que não podem se repetir dentro deste *grid*.

Por exemplo, imaginemos o Pedido de Vendas e não podemos permitir que o código do produto se repita, podemos definir no modelo este comportamento, sem precisar escrever nenhuma função específica para isso.

O método do modelo de dados *(Model)* que dever ser usado é o ***SetUniqueLine.***

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

No exemplo anterior o campo ***ZA2_AUTOR*** não poderá ter seu conteúdo repetido na *grid*. Também pode ser informado mais de um campo, criando assim um controle com chave composta.

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

No exemplo anterior a combinação do campo ***ZA2_AUTOR*** e ***ZA2_DATA*** não podem ter seu conteúdo repetido na *grid*.

Individualmente a repetição até poderá ocorrer, mas em conjunto não.

ZA2_AUTOR	ZA2_DATA	
001	01/01/11	Ok
001	02/01/11	Ok
002	02/01/11	Ok
001	01/01/11	Não permitido

8.3.4 Pré-validação de linha

Em um modelo de dados onde existam componentes de *grid* pode ser definida uma validação que será executada nas ações das linhas do *grid*. Podemos entender por essas ações a atribuição de valores, apagar ou recuperar uma linha.

Definimos a pré-validação de linha como um bloco de código no 4º parâmetros do método ***AddGrid.***

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

O bloco de código recebe como parâmetro:

- Um objeto que é a parte do modelo correspondente apenas ao *grid*;
- O número da linha;
- A ação executada:
  - ***SETVALUE*** – Para a atribuição de valores;
  - ***DELETE*** – Para deleção e recuperação da linha.

Campo onde se esta atribuíndo o valor, para deleção e recuperação da linha não é passado.

Esses parâmetros podem ser passados para a função que fará a validação.

A função chamada pelo bloco de código deve retornar um valor lógico, onde se .T. (verdadeiro) a troca de linha é realizada e .F. (falso) não é realizada.

Um exemplo da utilização da pré-validação de linha:

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

Local lRet       := .T.
Local oModel     := oModelGrid:GetModel()
Local nOperation := oModel:GetOperation()
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

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

Return lRet
```

No exemplo anterior não será permitida a deleção da linha na operação de alteração.

8.3.5 Validação da ativação do modelo (SetVldActivate)

É a validação realizada no momento da ativação do modelo, permitindo ou não a sua ativação.

Definimos a validação da ativação usando o método ***SetVldActivate.***

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

O bloco de código recebe como parâmetro um objeto que é o do modelo correspondente, porém, o modelo ainda não tem os dados carregados, pois a carga dos dados é feita após a sua ativação.

A função chamada pelo bloco de código deve retornar um valor lógico, onde se .T. (verdadeiro) a ativação é realizada e .F. (falso) não é realizada.

8.4 Manipulação da componente de grid

Veremos agora alguns tratamentos que podem ser feitos nos componentes de *grid* de um modelo de dados *(Model)*

8.4.1 Quantidade de linhas do componente de grid (Length)

Para se obter a quantidade de linhas do *grid* devemos utilizar o método ***Length***.

As linhas apagadas também são consideradas na contagem.

```
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
```

Se for passado um parâmetro no método ***Length***, o retorno será apenas a quantidade de linhas não apagadas da *grid*.

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

8.4.2 Ir para uma linha do componente de grid (GoLine)

Para movimentarmos o *grid*, ou seja, mudarmos a linha, onde o *grid* está posicionado, utilizamos o método ***GoLine*** passando como parâmetro o número da linha onde se deseja posicionar.

```
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 da linha de um componente de grid

Quando estamos falando do modelo de dados *(Model)* temos 3 operações básicas: **Inclusão, Alteração e Exclusão.**

Quando a operação é de inclusão, todos os componentes do modelo de dados *(Model)* estão incluídos, esse raciocínio também se aplica à exclusão, se esta é a operação, todos os componentes terão seus dados excluídos.

Porém, quando falamos da operação de alteração, não é bem assim.

Em um modelo de dados onde existam componentes do *grid*, na operação de alteração o *grid* pode ter linhas incluídas, alteradas ou excluídas, ou seja, o modelo de dados *(Model)* esta em alteração mas um *grid* pode ter tido as 3 operações em suas linhas.

Em *MVC* é possível saber que operações uma linha sofreu pelos seguintes métodos de *status*:

IsDeleted: Informa se uma linha foi apagada. Retornando .T. (verdadeiro) a linha foi apagada.

IsUpdated: Informa se uma linha foi alterada. Retornando .T. (verdadeiro) a linha foi alterada.

IsInserted: Informa se uma linha foi inserida, ou seja, se é uma linha nova na *grid*. Retornando .T. (verdadeiro) a linha foi inserida.

Exemplo:

```
Static Function COMP23ACAO()

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


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
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)
```

Mas de um método de status pode retornar .T. (verdadeiro) para a mesma linha. Se uma linha foi incluída, o ***IsInserted*** retornará .T. (verdadeiro), em seguida ela foi alterada, o ***IsUpdated*** retornará .T. (verdadeiro) e em seguida a mesma linha foi apagada, ***IsDeleted*** também retornará .T. (verdadeiro).

8.4.4 Adição uma linha a grid (AddLine)

Para adicionarmos uma linha a um componente do *grid* do modelo de dados *(Model)* utilizamos o método ***AddLine.***

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

O método AddLine retorna a quantidade total de linhas da *grid*. Se a *grid* já possui 2 linhas e tudo correu bem na adição da linha, o AddLine retornara 3, se ocorreu algum problema retornará 2, pois a nova linha não foi inserida.

Os motivos para a inserção não ser bem sucedida poderá ser algum campo obrigatório não informado, a pós-validação da linha retornou .F. (falso), atingiu a quantidade máxima de linhas

para o *grid*, por exemplo.

8.4.5 Apagando e recuperando uma linha da grid (DeleteLine e UnDeleteLine)

Para apagarmos uma linha de um componente de *grid* do modelo de dados *(Model)* utilizamos o método ***DeleteLine.***

```
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
```

O método ***DeleteLine*** retorna .T. (verdadeiro) se a deleção foi bem sucedida. Um motivo para que não seja é a pré-validação da linha retornar .F. (falso).

Se quisermos recuperar uma linha da *grid* que está apagada utilizamos o método ***UnDeleteLine.***

```
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
```

O método ***UnDeleteLine*** retorna .T. (verdadeiro) se a recuperação foi bem sucedida. Um motivo para que não seja é a pré-validação da linha retornar .F. (falso).

8.4.6 Permissões para uma grid

Se quisermos limitar que uma linha da *grid* possa ser inserida, alterada ou apagada, para fazermos uma consulta, por exemplo, utilizamos um dos métodos abaixo:

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

***SetNoInsertLine*:** Não permitir serem inseridas linhas na *grid*.

Exemplo:

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

SetNoUpdateLine: Não permite alterar as linhas do *grid*.

Exemplo:

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

SetNoDeleteLine: Não permite apagar linhas do *grid*.

Exemplo:

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

Esses métodos podem ser informados no momento da definição do modelo de dados (*Model*).

8.4.7 Permissão de grid sem dados (SetOptional)

Por padrão, quando temos um modelo de dados *(Model)* onde há um componente de *grid*, deve ser informada pelo menos uma linha neste *grid*.

Mas vamos imaginar um modelo onde tivéssemos o cadastro de produtos e seus acessórios. É um modelo *Master-Detail*, teremos para cada produto ***n*** acessórios, mas também teremos produtos que não têm acessório algum. Assim, esta regra de que deve haver pelo menos uma linha informada na *grid* não pode ser aplicada.

Neste caso utilizamos o método ***SetOptional*** para permitir que o *grid* tenha ou não pelo menos uma linha digitada, ou seja, para dizer que a digitação de dados do *grid* é opcional.

Esse método deve ser informado ao definir o modelo de dados (*Model*).

Exemplo:

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

Se um *grid* for opcional e na estrutura houver campos obrigatórios, só será validado se estes campos foram informados e somente se a linha sofrer alguma alteração em qualquer campo.

O método ***IsOptional*** pode ser utilizado para saber se o componente de *grid* tem ou não esta característica. Se retornar .T. (verdadeiro) o componente permite que não existam linhas digitadas. Este método pode ser útil em validações.

8.4.8 Guardando e restaurando o posicionamento do grid (FWSaveRows / FWRestRows)

Um cuidado que devemos ter quando escrevemos uma função, mesmo que não seja para uso em *MVC*, é restaurarmos as áreas das tabelas que desposicionamos.

Analogamente, devemos ter o mesmo cuidado para os componentes do *grid* que desposicionamos em uma função, com o uso do método ***GoLine***, por exemplo.

Para isso utilizaremos as funções ***FWSaveRows*** para salvar o posicionamento das linhas dos *grid*s do modelo de dados *(Model)* e o ***FWRestRows*** para restaurar esses posicionamentos.

Exemplo:

```
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 )
```

Obs.: O ***FWSaveRows*** guarda o posicionamento de todos os *grid*s do modelo de dados *(Model)* e o ***FWSaveRows*** restaura o posicionamento de todos os *grid*s do *model*.

8.4.9 Definição da quantidade máxima de linhas do grid (SetMaxLine)

Por padrão a quantidade máxima de linhas de um componente de *grid* é 990.

Se for necessário alterar esta quantidade utiliza-se o método ***SetMaxLine***. Este método deve ser usado na definição do modelo de dados (*Model*), ou seja, na ***ModelDef***.

Importante: A quantidade se refere sempre ao total de linhas, independentemente se estas estão apagadas ou não.

8.5 Obtenção e atribuição de valores ao modelo de dados

As operações mais comuns que faremos em um modelo de dados *(Model)* é obter e atribuir valores.

Para isso utilizamos um dos métodos abaixo:

GetValue: Obtém um dado do modelo de dados *(Model)*. Podemos obter o dado a partir do modelo completo ou a partir de um componente dele.

A partir do modelo de dados *(Model)* completo.

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

Onde ***ZA1MASTER*** é o identificador *(ID)* do componente e ***ZA1_MUSICA*** é o campo do qual se deseja obter o dado.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Ou a partir de um componente do modelo de dados *(Model)*.

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

SetValue: Atribui um dado ao modelo de dados *(Model)*. Podemos atribuir o dado a partir do modelo completo ou a partir de uma parte dele.

A partir do modelo de dados *(Model)* completo

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

Onde ***ZA1MASTER*** é o identificador *(ID)* do componente e ***ZA1_MUSICA*** é o campo no qual se deseja atribuir o dado e ***000001*** é o dado que se deseja atribuir.

Ou a partir de um componente do modelo de dados *(Model).*

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

Quando utilizamos o **SetValue** para atribuir um dado a um campo as validações deste campo são executadas e também são disparados os seus gatilhos.

O **SetValue** retorna .T. (verdadeiro) se a atribuição foi bem sucedida, os motivos para que não seja podem ser que o dado não satisfez a validação ou o modo de edição (***WHEN***) não foi satisfeito, etc.

LoadValue: Atribui um dado ao modelo de dados *(Model)*. Podemos atribuir o dado a partir do modelo completo ou a partir de uma parte dele.

A partir do modelo de dados *(Model)* completo

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

Onde ***ZA1MASTER*** é o identificador *(ID)* do componente e ***ZA1_MUSICA*** é o campo onde se deseja atribuir o dado e ***000001*** é o dado que se deseja atribuir.

Ou a partir de um componente do modelo de dados *(Model)*.

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

A diferença entre o ***LoadValue*** e o ***SetValue*** é que o ***LoadValue*** não executa as validações nem dispara os gatilhos do campo. Ele força a atribuição de dado.

Importante: Utilize sempre o ***SetValue*** para atribuir um dado, evite o ***LoadValue***. Só o utilize quando for extremamente necessário.

8.6 Comportamento

Veremos como alterar alguns dos comportamentos padrões do modelo de dados *(Model).*

8.6.1 Alteração de dados de um componente no modelo de dados (SetOnlyView)

Se quisermos que um determinado componente do modelo de dados *(Model)* não permita alteração em seus dados, que seja apenas para visualização, utilizamos o método ***SetOnlyView.***

Esse método deve ser informado no momento da definição do Model.

Exemplo:

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

8.6.2 Não gravar dados de um componente do modelo de dados (SetOnlyQuery)

A persistência dos dados (gravação) é feita automaticamente pelo modelo de dados *(Model).*

Se quisermos que um determinado componente do modelo de dados *(Model)* permita inclusão e/ou alteração em seus dados, porém, que estes dados não sejam gravados, utilizamos o método ***SetOnlyQuery.***

Esse método deve ser informado no momento da definição do Model.

Exemplo:

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

8.6.3 Obtenção da operação que está sendo realizada (GetOperation)

Para sabermos a operação com que um modelo de dados *(Model)* está trabalhando, usamos o método ***GetOperation.***

Esse método retorna:

- O valor 3 quando é uma **inclusão;**
- O valor 4 quando é uma **alteração**;
- O valor 5 quando é uma **exclusão.**

```
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
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
        lRet := .F.
        Help( ,, 'Help',, 'Não permitido apagar linhas na alteração.' + CRLF + ;
        'Você esta na linha ' + Alltrim( Str( nLinha ) ), 1, 0 )
    EndIf

    Return lRet
```

No *MVC* foram criadas várias diretivas de compilação ***#DEFINE*** para facilitar o desenvolvimento e tornar a leitura de uma aplicação mais fácil.

Para utilizar este ***#DEFINE*** é preciso incluir a seguinte diretiva no fonte:

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

Para as operações do modelo de dados *(Model)* podem ser utilizados:

- ***MODEL_OPERATION_INSERT*** para **inclusão**;
- ***MODEL_OPERATION_UPDATE*** para **alteração**;
- ***MODEL_OPERATION_DELETE*** para **exclusão**.

Assim no exemplo dado acima podemos escrever:

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

8.6.4 Gravação manual de dados (FWFormCommit)

A gravação dos dados do modelo de dados *(Model)* (persistência) é realizada pelo *MVC* onde são gravados todos os dados das entidades do *model*.

Porém, pode haver a necessidade de se efetuar gravações em outras entidades que não participam do modelo. Por exemplo, quando incluímos um Pedido de Vendas é preciso atualizar o valor de pedidos em aberto do Cadastro de Clientes. O cabeçalho e itens do pedido fazem parte do modelo e serão gravados, o cadastro de Cliente não faz parte, mas precisa ser atualizado também.

Para este tipo de situação é possível intervir no momento da gravação dos dados.

Para isso definimos um bloco de código no 4º. parâmetro da classe de construção do modelo de dados *(Model)* ***MPFormModel.***

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

O bloco de código recebe como parâmetro um objeto que é o modelo e que pode ser passado à função que fará a gravação.

Diferentemente dos blocos de código definidos no modelo de dados *(Model)* para validação que complementam a validações feitas pelo *MVC*, o bloco de código para gravação substitui a gravação dos dados. Então ao ser definido um bloco de código para gravação, passa ser responsabilidade da função criada, a gravação de todos os dados inclusive os dados do modelo de dados em uso.

Para facilitar o desenvolvimento foi criada a função ***FWFormCommit*** que fará a gravação dos dados do objeto de modelo de dados *(Model)* informado.

```
Static Function COMP011GRV ( oModel )


FWFormCommit( oModel )


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

Importante: Não devem ser feitas atribuições de dados no modelo *(Model)* dentro da função de gravação. Conceitualmente ao se iniciar a gravação, o modelo de dados *(Model)* já passou por toda a validação, ao tentar atribuir um valor, esse valor pode não satisfazer a validação do campo tornando o modelo de dados *(Model)* invalidado novamente e o que ocorrerá é a gravação de dados inconsistentes.

8.7 Regras de preenchimento (AddRules)

Uma nova característica que foi implementada no *MVC* são as regras de preenchimento, onde o preenchimento de um campo depende do preenchimento de outro.

Por exemplo, podemos definir que o campo Código da Loja de uma entidade, só pode ser preenchido após o preenchimento do campo Código do Cliente.

As regras de preenchimento podem ser de 3 tipos:

- **Tipo 1 Pré-Validação**

  Adiciona uma relação de dependência entre campos do formulário, impedindo a atribuição de valor caso os campos de dependência não tenham valor atribuído. Por exemplo, o preenchimento do campo Código da Loja só pode ser preenchido após o preenchimento do campo Código do Cliente.

- **Tipo 2 Pos-Validação**

  Adiciona uma relação de dependência entre a referência de origem e destino, provocando uma reavaliação do destino em caso de atualização da origem. Por exemplo, após o preenchimento do campo Código da Loja a validação é reavaliado caso o Código do Cliente. seja alterado.

- **Tipo 3 Pré e Pós-Validação**

  São os tipos 1 e 2 simultaneamente

Exemplo:

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

O ***ZA3MASTER*** é o identificador *(ID)* do componente do modelo de dados *(Model)* onde esta o campo de destino, ***ZA3_LOJA*** é o campo destino, o segundo ***ZA3MASTER*** é do componente do modelo de dados *(Model)* onde está o campo de origem, ***ZA3_DATA*** é o campo de origem.

9.Tratamentos de *interface*

Veremos alguns tratamentos que podem ser feitos na *interface (View)* conforme a necessidade.

- Criação de botões;
- Criação de pastas;
- Agrupamento de campos;
- Incremento de campos;
- Etc.

9.1 Campo Incremental (AddIncrementField)

Podemos fazer com que um campo do modelo de dados (*Model*) que faça parte de um componente de *grid,* possa ser incrementado unitariamente a cada nova linha inserida.

Por exemplo, imaginemos o Pedido de Vendas, nos itens, o número do item pode ser um campo incremental.

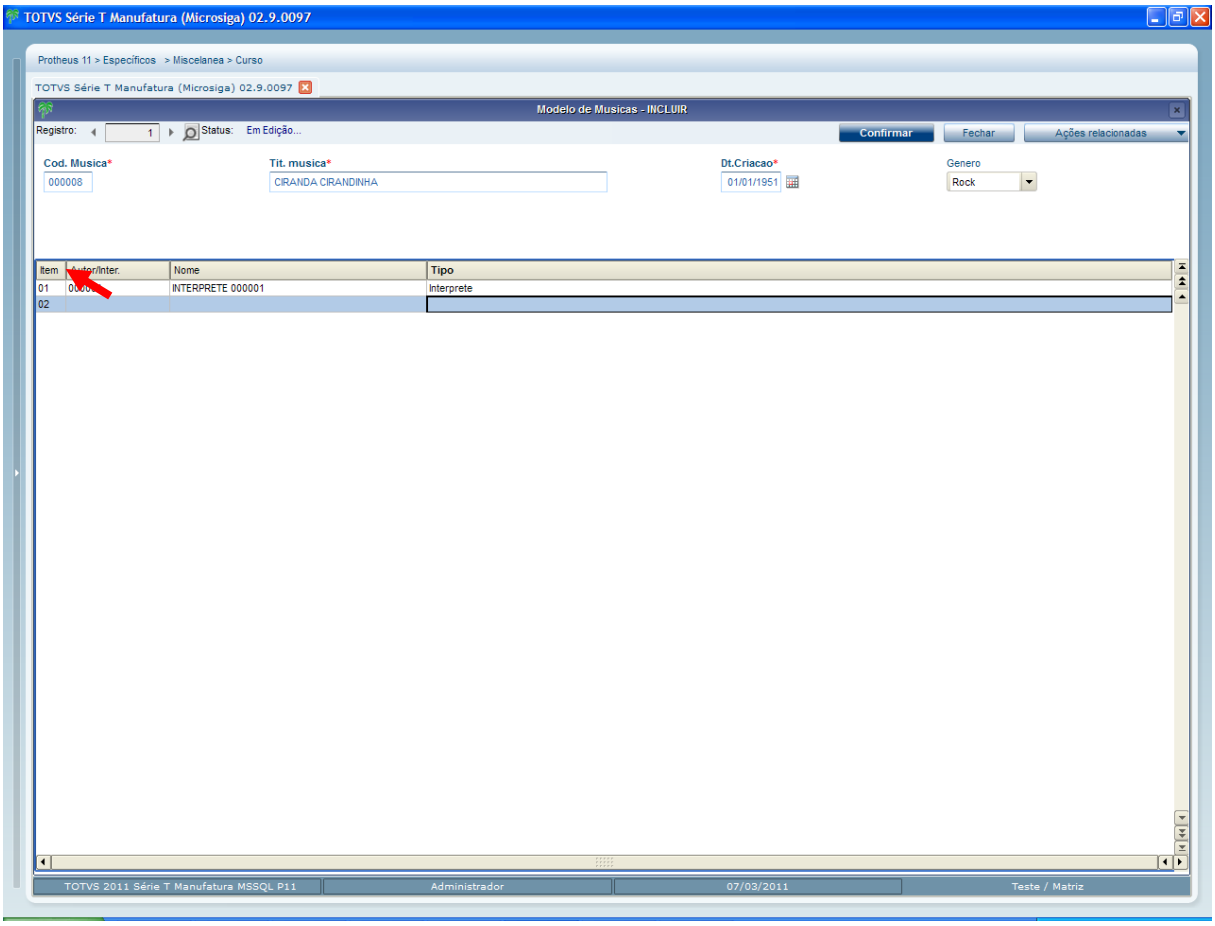
Para isso utilizamos o método ***AddIncrementField.***

Exemplo:

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

Onde ***VIEW_ZA2*** é o identificador *(ID)* do componente da *interface (View)*, onde se encontra o campo e ***ZA2_ITEM*** o nome do campo que será incrementado.

Visualmente temos :



Importante: Esse comportamento só acontece quando a aplicação está sendo usada por sua *interface (View)*. Quando o modelo de dados é usado diretamente (*Web Services*, rotinas automática, etc.) o campo incremental tem que ser informado normalmente.

9.2 Criação de botões na barra de botões (AddUserButton)

Para a criação de botões adicionais na barra de botões da *interface* utilizamos o método ***AddUserButton.***

Exemplo:

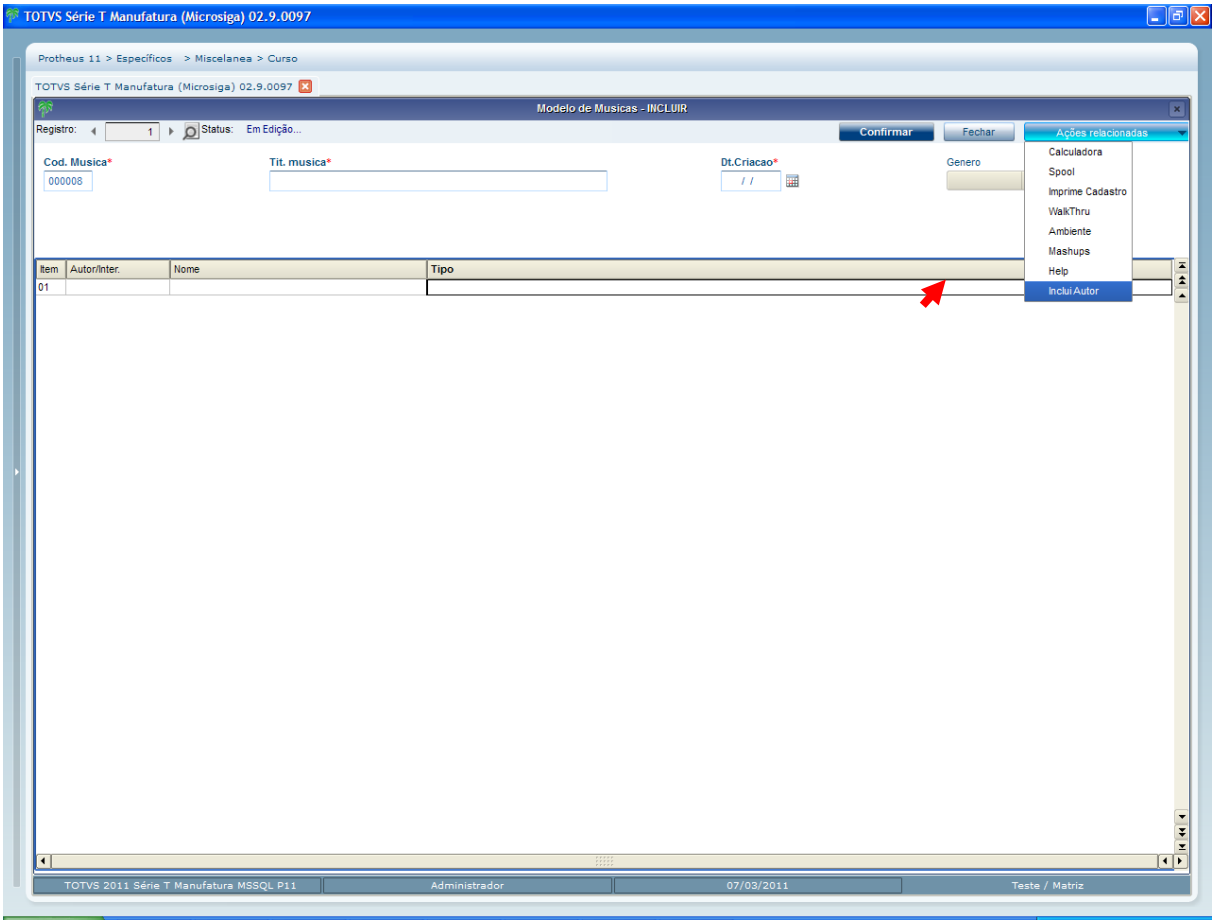
```
oView:AddUserButton( 'Inclui Autor', 'CLIPS', {|oView| COMP021BUT()} )
```

Onde o **Inclui Autor**, é o texto que será apresentado no botão, **CLIPS** é o nome da imagem do RPO[2] que será usada para o botão e o 3º parâmetro é o bloco de código que será executado ao acionar o botão.

[2] RPO – Repositório do Microsiga Protheus para aplicações e imagens

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Visualmente temos:



9.3 Título do componente (EnableTitleView)

No *MVC* podemos atribuir um título para identificar cada componente da *interface*, para isso usamos o método ***EnableTitleView.***

Exemplo:

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

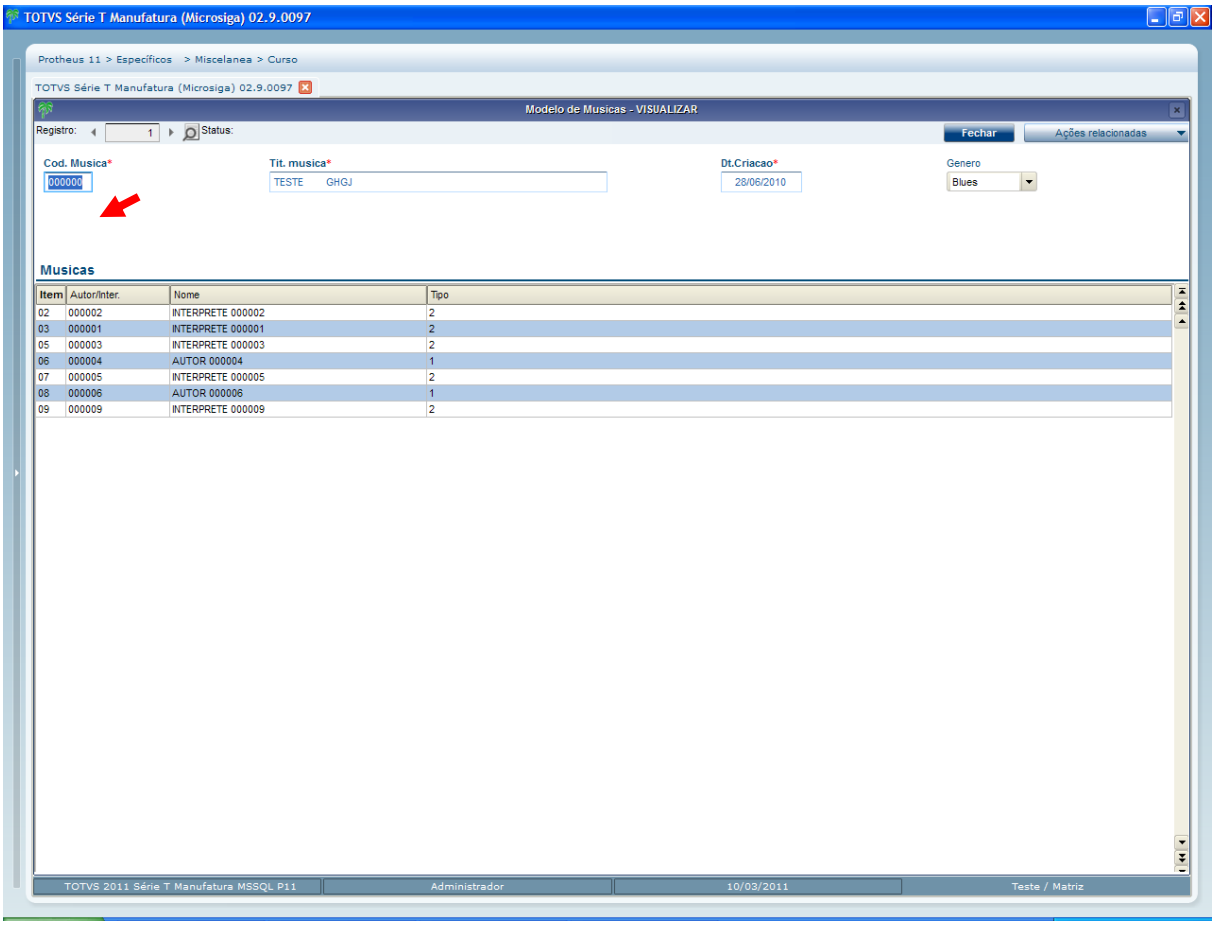
Onde ***VIEW_ZA2*** é o identificador *(ID)* do componente da *interface (View)*, e ***'MUSICAS'*** o titulo que deseja para o componente

Podemos ainda usar:

```
oView:EnableTitleView('VIEW_ZA2')
```

Onde o titulo que será exibido é o que foi definido no método ***SetDescription*** do modelo de dados *(Model)* para o componente.

Visualmente temos:



9.4 Edição de Campos no componente de grid (SetViewProperty)

Uma nova característica que o *MVC* possui no uso da *interface*, é para um componente de *grid*, fazer ao mesmo tempo a edição de dados diretamente na *grid* e/ou em uma tela no layout de formulário.

Para isso utilizamos o método ***SetViewProperty.*** Esse método habilita alguns comportamentos específicos ao componente da *interface (View)*, conforme a diretiva recebida.

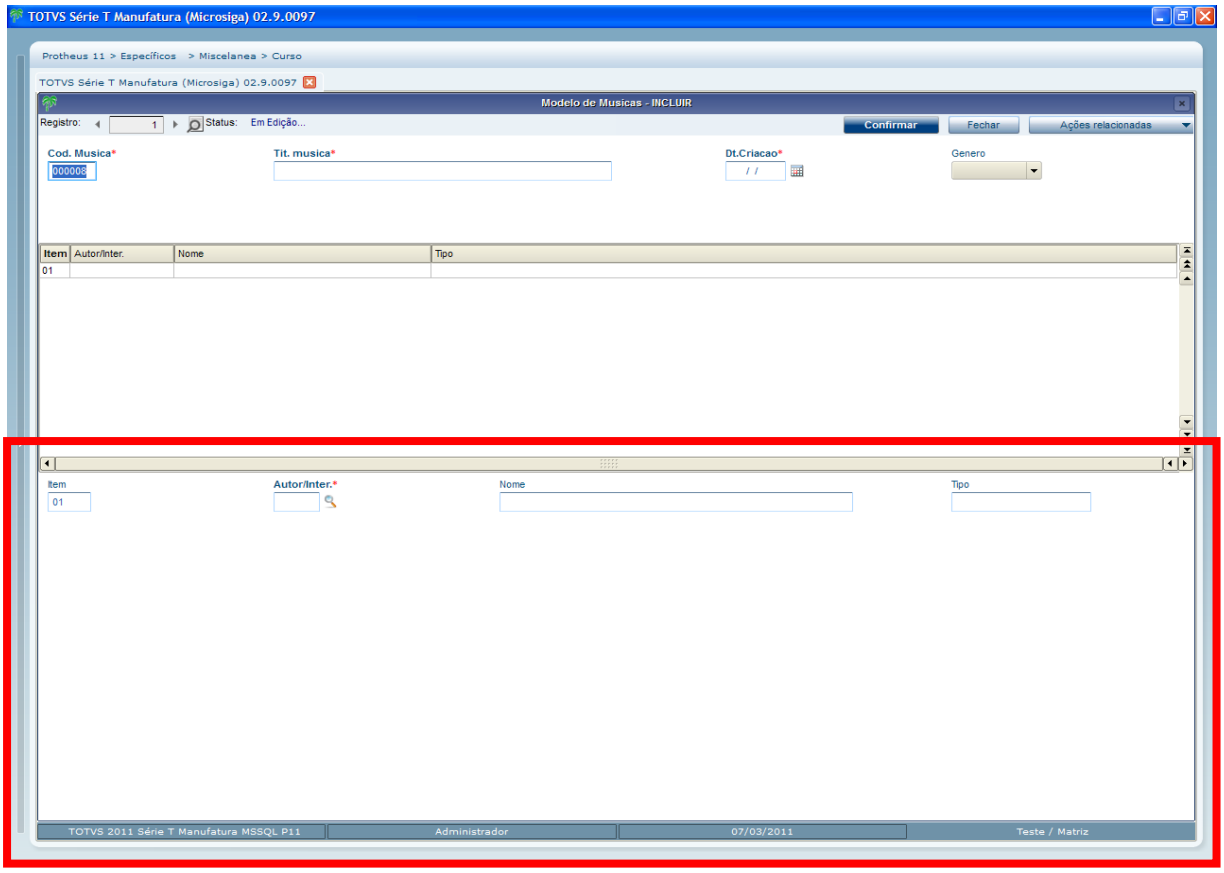
Exemplo:

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

Onde ***VIEW_ZA2*** é o identificador *(ID)* do componente da *interface (View)*, onde se encontra o campo e ***ENABLEDGRIDDETAIL*** é a diretiva que habilita o comportamento.

{60} é o percentual que o formulário de edição ocupará do tamanho que o componente de *grid* ocupa atualmente. Exemplificando numericamente, se para o componente de *grid* foi definido que ele utilizará 50% da tela, ao se colocar 60 (60%) no parâmetro, quer se indicar que dos 50% destinados ao componente de *grid*, 60% serão usados para o formulário de edição.

Visualmente temos:



9.5 Criação de pastas (CreateFolder)

Em *MVC* podemos criar pastas onde serão colocados os componentes da *interface (View).*

Para isso utilizamos o método ***CreateFolder.***

Exemplo:

```
oView:CreateFolder( 'PASTAS' )
```

Devemos dar um identificador *(ID)* para cada componente da *interface (View)*. ***PASTAS*** é o identificador *(ID)* dado às pastas.

Após a criação da pasta principal, precisamos criar as abas desta pasta. Para isso é usado o método ***AddSheet.***

Por exemplo, criaremos 2 abas:

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

Onde ***PASTAS*** é o identificador *(ID)* da pasta, e ***ABA01*** e ***ABA02*** são os IDs dados a cada aba e ***Cabeçalho*** e ***Item*** são os títulos de cada aba.

Para que possamos colocar um componente em uma aba, precisamos criar um box, um objeto, para receber os elementos da *interface (View)*.

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

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

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

Devemos dar um identificador *(ID)* para cada componente da *interface (View)*.

- ***SUPERIOR*** e ***INFERIOR*** são os *IDs* dados a cada ***box.***
- ***100*** indica o percentual que o ***box*** ocupará da aba.
- ***PASTAS*** é o identificador *(ID)* da pasta.
- ***ABA01*** e ***ABA02*** os *IDs* das abas.

Precisamos relacionar o componente da *interface (View)* com um **box** para exibição, para isso usamos o método ***SetOwnerView.***

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

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

Resumindo:

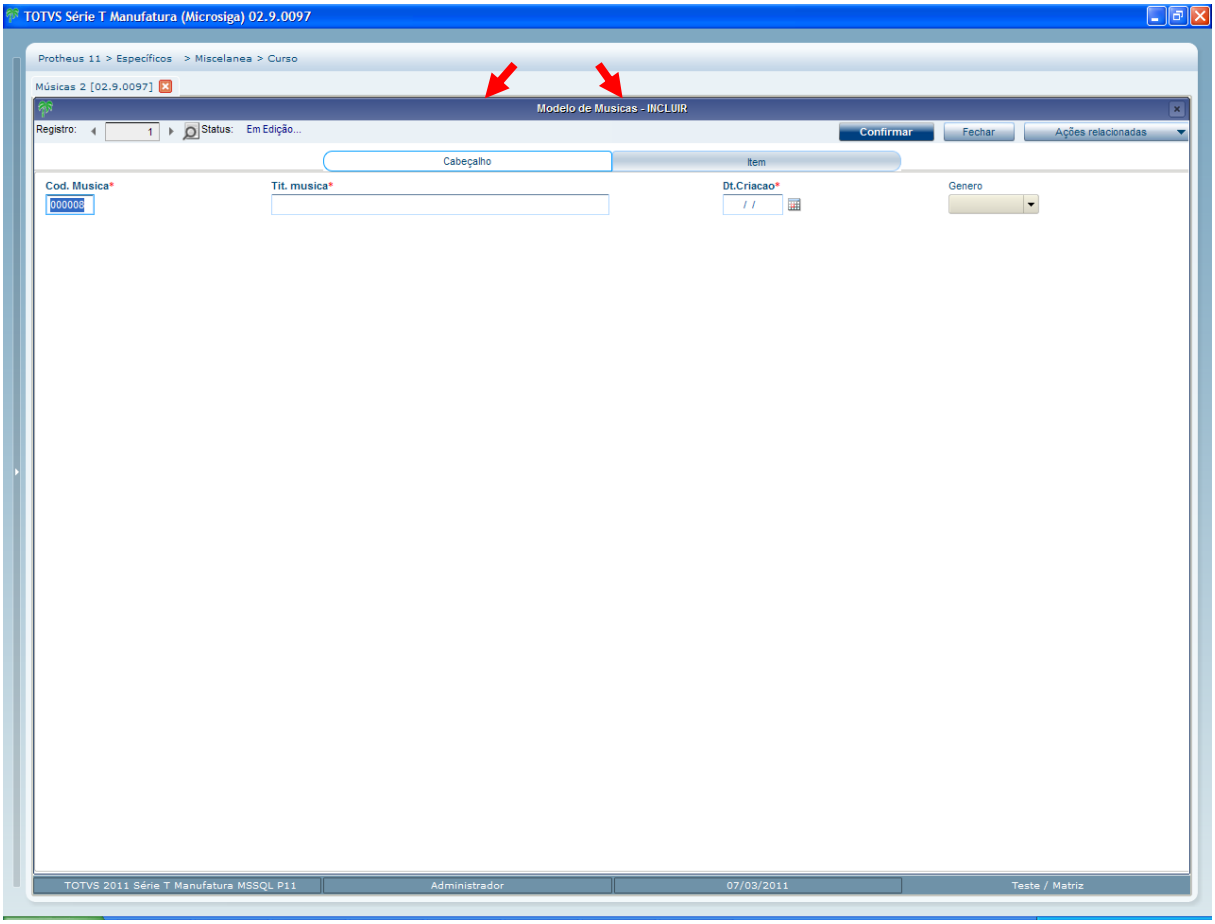
```
// Cria Folder na view

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'  )
```

Quando as pastas são definidas utilizando os metadados (dicionários), automaticamente a

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

interface (View) cria estas pastas. Se o componente colocado em uma das abas possui pastas definidas no metadados, estas pastas serão criadas dentro da aba onde ele se encontra.

Visualmente temos:



9.6 Agrupamento de campos (AddGroup)

Uma nova característica que o *MVC* possui para uso da *interface* é para um componente de formulário, fazer o agrupamento dos campos na tela.

Por exemplo, em um cadastro de clientes podemos ter campos para o endereço de entrega, correspondência e faturamento. Para uma visualização melhor poderíamos agrupar os campos de cada endereço.

Para isso usamos o método ***AddGroup.***

Exemplo:

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

Devemos dar um identificador *(ID)* para cada componente da *interface (View)*.

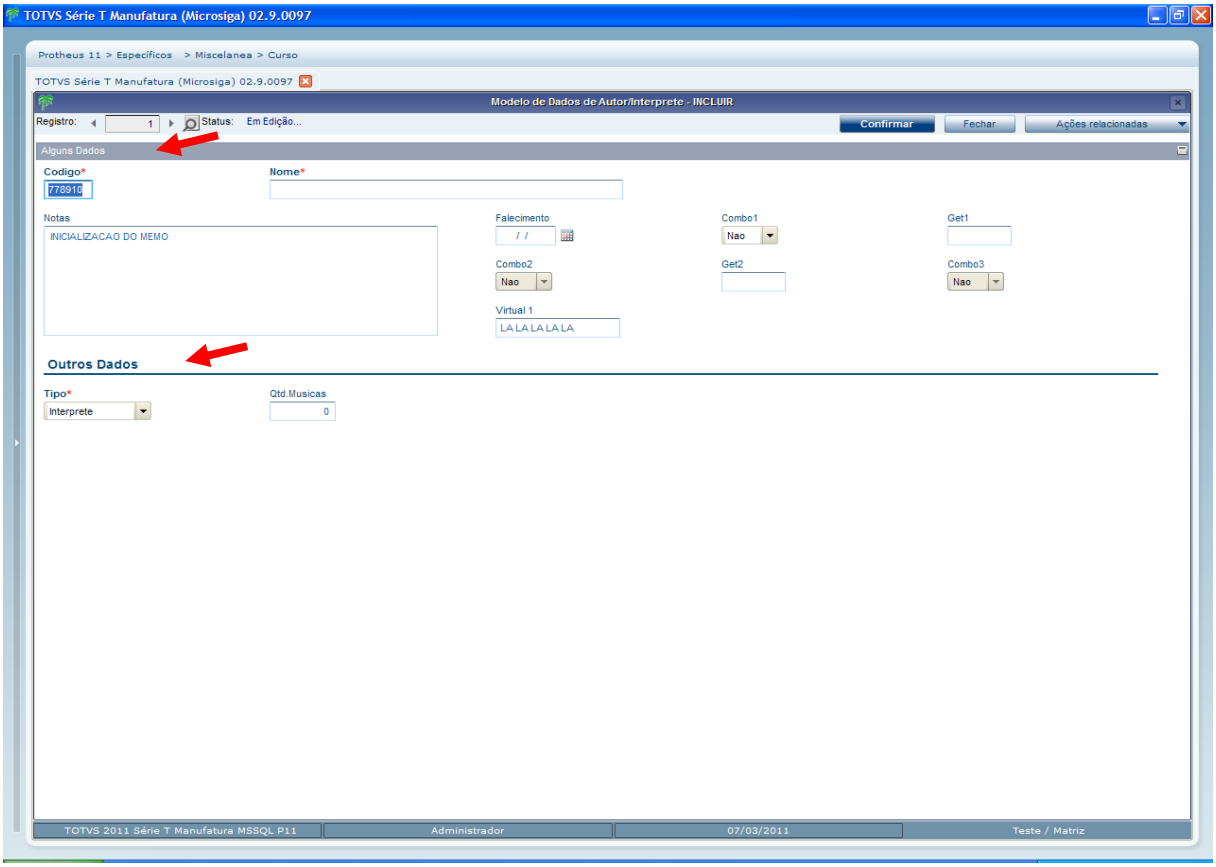
GRUPO01 é o identificador *(ID)* dado ao agrupamento, o 2º parâmetro é o título que será

apresentado no agrupamento e 1 é o tipo de agrupamento, podendo ser 1- Janela; 2 - Separador

Com o agrupamento criado, precisamos dizer quais campos farão parte deste agrupamento. Para isso alteraremos uma propriedade da estrutura de alguns campos. Usaremos o método ***SetProperty***, que pode ser visto mais detalhadamente no capítulo 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' )
```

Visualmente temos:



Observação: Os agrupamentos serão exibidos na *interface (View)* na ordem de sua criação.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

9.7 Ação de interface (SetViewAction)

Existe no *MVC* a possibilidade de se executar uma função em algumas ações da *interface (View)*. Esse recurso pode ser usado quando queremos executar algo na *interface* e que não tem reflexo no modelo de dados *(Model)* como um ***Refresh*** de tela por exemplo.

Isso é possível nas seguintes ações:

- *Refresh* da interface;
- Acionamento do botão confirmar da *interface*;
- Acionamento do botão cancelar da *interface*;
- Deleção da linha da *grid;*
- Restauração da linha da *grid;*

Para isso usamos o método na ***SetViewAction***

A sua sintaxe é:

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

Onde:

cActionlID ID do ponto onde a ação será executada que podem ser:

REFRESH	Executa a ação no ***Refresh*** da View;
BUTTONOK	Executa a ação no acionamento do botão confirmar da View;
BUTTONCANCEL	Executa a ação no acionamento do botão cancelar da View;
DELETELINE	Executa a ação na deleção da linha da *grid;*
UNDELETELINE	Executa a ação na restauração da linha da *grid;*

bAction Bloco com a ação a ser executada. Recebe como parâmetro:

REFRESH	Recebe como parâmetro o objeto de View;
BUTTONOK	Recebe como parâmetro o objeto de View;
BUTTONCANCEL	Recebe como parâmetro o objeto de View;
DELETELINE	Recebe como parâmetro o objeto de View, identificador *(ID)* da View e número da linha.
UNDELETELINE	Recebe como parâmetro o objeto de View, identificador *(ID)* da View e número da linha

Exemplo:

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

Importante: Essas ações são executadas apenas quando existe uma *interface (View)*. O que não ocorre quando temos o instânciamento direto do modelo, rotina automática ou *Web Service*. Deve-se evitar então colocar nestas funções ações que possam influenciar a regra de negócio, pois na execução da aplicação sem *interface* essas ações não serão executadas.

9.8 Ação de interface do campo (SetFieldAction)

Existe no *MVC* a possibilidade de se executar uma função após a validação de campo de algum componente do modelo de dados *(Model).*

Esse recurso pode ser usado quando queremos executar algo na *interface* e que não tem reflexo no modelo, como um ***Refresh*** de tela ou abrir uma tela auxiliar, por exemplo.

Para isso usamos o método na ***SetFieldAction.***

A sua sintaxe é:

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

Onde:

cIDField ID do campo (nome):

bAction Bloco com a ação a ser executada, recebe como parâmetro:

- Objeto De View
- O identificador *(ID)* Da View
- O identificador *(ID)* Do Campo
- Conteúdo Do Campo

Exemplo:

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

Importante:

- Essas ações são executadas após a validação do campo.
- Essas ações são executadas apenas quando existe uma *interface (View)*. O que não ocorre quando temos o instânciamento direto do modelo, rotina automática ou *Web Service*.
- Deve-se evitar então colocar nestas funções ações que possam influenciar a regra de negócio, pois na execução da aplicação sem *interface* essas ações não serão executadas.

9.9 Outros objetos (AddOtherObjects)

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Na construção de algumas aplicações pode ser que tenhamos que adicionar à *interface* um componente que não faz parte da *interface* padrão do *MVC*, como um gráfico, um calendário, etc.

Para isso usaremos o método ***AddOtherObject***

Sua sintaxe é :

AddOtherObject(<Id>, <Code Block a ser executado>)

Onde o 1º parâmetro é o identificador *(ID)* do ***AddOtherObjects*** e 2º parâmetro o código de bloco que será executado para a criação dos outro objeto.

O *MVC* se limita a fazer a chamada da função, a responsabilidade de construção e atualização dos dados cabe ao desenvolvedor em sua função.

Exemplo:

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

Note que o 2º parâmetro recebe como parâmetro um objeto que é o ***container*** onde o desenvolvedor deve colocar seus objetos.

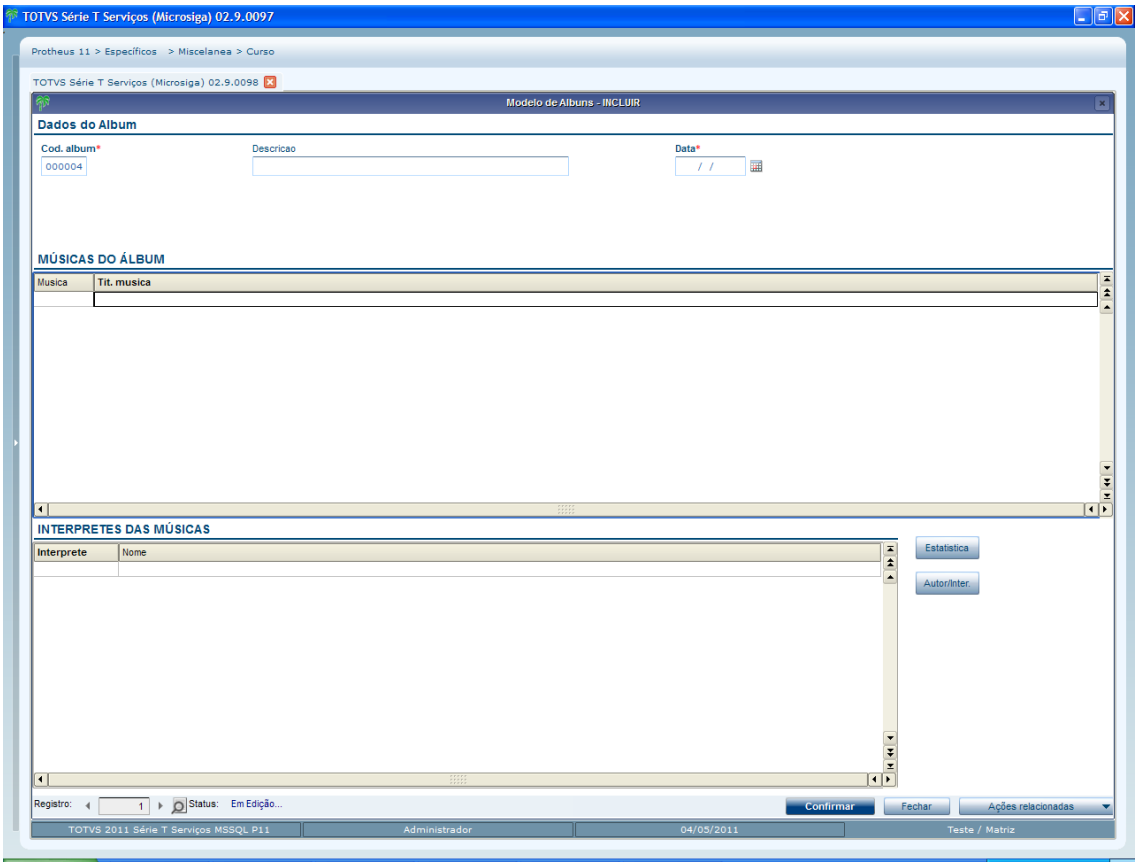
Abaixo segue um exemplo do uso do método, onde colocamos em pedaço da *interface (View)* 2 botões. Observe os comentários no fonte:

```
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
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Visualmente temos:



10.Tratamentos de estrutura de dados

Como dito anteriormente o *MVC* não trabalha vinculado aos metadados do Microsiga Protheus, ele trabalha vinculado a estruturas. Essas estruturas, por sua vez é que podem ser construídas a partir dos metadados (dicionários).

Veremos alguns tratamentos que podem ser feitos nas estruturas conforme a necessidade.

10.1 Seleção de campos para a estrutura (FWFormStruct)

Ao criarmos uma estrutura baseada no metadados (dicionários), utilizando a função ***FWFormStruct***, ela leva em consideração todos os campos da entidade, respeitando nível, módulo, uso , etc.

Se quisermos selecionar quais os campos do metadados (dicionários) que farão parte da estrutura, devemos utilizar o 3º parâmetro da ***FWFormStruct***, que é um bloco de código que será executada para cada campo que a função trouxer do metadados (dicionários) e que recebe como parâmetro o nome do campo.

O bloco de código deve retornar um valor lógico, onde se .T. (verdadeiro) o campo fará parte da estrutura, se .F. (falso) não fará.

Exemplo:

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

Onde a função pode ser:

```
Static Function COMP11STRU( cCampo )
Local lRet := .T.

If cCampo == 'ZA0_QTDMUS'
	lRet := .F.
EndIf

Return lRet
```

No exemplo de função dado o campo ***ZA0_QTDMUS*** não fará parte da estrutura.

O dicionário de campos (SX3) do metadado é posicionado para cada campo.

Importante: Esse tratamento pode ser feito tanto para as estruturas que serão usadas no modelo de dados *(Model)* quanto na *interface (View).*

Mas tome o seguinte cuidado: Se for removido da estrutura da *interface (View)* um campo obrigatório, mesmo ele não sendo exibido para o usuário, o modelo de dados *(Model)* fará a sua validação dizendo que um campo obrigatório não foi preenchido.

10.2 Remoção de campos da estrutura (RemoveField)

Uma forma para a retirada de um campo da estrutura é o uso do método ***RemoveField.***

Exemplo:

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


oStruZA0: RemoveField('ZA0_QTDMUS')
```

No exemplo acima o campo ***ZA0_QTDMUS*** foi removido da estrutura.

Importante: Esse tratamento pode ser feito tanto para as estruturas que serão usadas no modelo de dados *(Model)* quanto na *interface (View)*.

Mas tome o seguinte cuidado: Se for removido da estrutura da *interface (View)* um campo obrigatório, mesmo ele não sendo exibido para o usuário, o modelo de dados *(Model)* fará a sua validação dizendo que um campo obrigatório não foi preenchido.

10.3 Alteração de propriedades do campo (SetProperty)

Ao criarmos uma estrutura baseada no metadados (dicionários), utilizando a função ***FWFormStruct***, são respeitadas as propriedades que o campo tem como validação, inicializador padrão e modo de edição, etc.

Se houver a necessidade de mudar alguma propriedade do campo na estrutura, usaremos o método ***SetProperty*** para isso.

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

Onde o 1º parâmetro é o nome do campo que se deseja mudar ou atribuir a propriedade o 2º é a propriedade que esta sendo tratada e o 3º é o valor para a propriedade.

No exemplo anterior o campo ***ZA0_QTDMUS*** só poderá ser editado na operação de inclusão.

As propriedades para os campos da estrutura do modelo de dados *(Model)* são:

Propriedade para campos da estrutura do modelo de dados (*Model*)	**Tipo**	**Descrição**
MODEL_FIELD_TITULO	C	Titulo
MODEL_FIELD_TOOLTIP	C	Descrição completa do campo
MODEL_FIELD_IDFIELD	C	Nome *(ID)*
MODEL_FIELD_TIPO	C	Tipo
MODEL_FIELD_TAMANHO	N	Tamanho
MODEL_FIELD_DECIMAL	N	Decimais
MODEL_FIELD_VALID	b	Validação
MODEL_FIELD_WHEN	B	Modo de edição
MODEL_FIELD_VALUES	A	Lista de valores permitido do campo (combo)
MODEL_FIELD_OBRIGAT	L	Indica se o campo tem preenchimento obrigatório
MODEL_FIELD_INIT	B	Inicializador padrão
MODEL_FIELD_KEY	L	Indica se o campo é chave
MODEL_FIELD_NOUPD	L	Indica se o campo pode receber valor em uma operação de *update*.
MODEL_FIELD_VIRTUAL	L	Indica se o campo é virtual

As propriedades para os campos da estrutura da *interface (View)* são:

Propriedade para campos da estrutura da *interface* (*View*)	**Tipo**	**Descrição**
MVC_VIEW_IDFIELD	C	Nome do Campo
MVC_VIEW_ORDEM	C	Ordem
MVC_VIEW_TITULO	C	Titulo do campo
MVC_VIEW_DESCR	C	Descrição do campo
MVC_VIEW_HELP	A	Array com Help
MVC_VIEW_PICT	C	Picture
MVC_VIEW_PVAR	B	Bloco de Picture Var
MVC_VIEW_LOOKUP	C	Consulta F3
MVC_VIEW_CANCHANGE	L	Indica se o campo é editável
MVC_VIEW_FOLDER_NUMBER	C	Pasta do campo
MVC_VIEW_GROUP_NUMBER	C	Agrupamento do campo
MVC_VIEW_COMBOBOX	A	Lista de valores permitido do campo (Combo)
MVC_VIEW_MAXTAMCMB	N	Tamanho Maximo da maior opção do combo
MVC_VIEW_INIBROW	C	Inicializador de *Browse*
MVC_VIEW_VIRTUAL	L	Indica se o campo é virtual
MVC_VIEW_PICTVAR	C	Picture Variável

Os nomes de propriedades citados nas tabelas dadas são na verdade diretivas de compilação do tipo ***#DEFINE***.

Para utilizar este ***#DEFINE*** é preciso incluir a seguinte diretiva no fonte:

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

Também é possível atribuirmos uma propriedade para todos os campos da estrutura de só vez utilizando no nome do campo o asterisco "*"

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

10.4 Criação de campos adicionais na estrutura (AddField)

Se quisermos criar um campo em uma estrutura já existente, utilizamos o método ***Addfield.***

Há diferenças na seqüência de parâmetros deste método para adicionar campos para a estrutura do modelo de dados *(Model)* ou da *interface (View).*

Sua sintaxe para o modelo de dados *(Model)* é:

AddField (cTitulo, cTooltip, cIdField, cTipo, nTamanho, nDecimal, bValid, bWhen, aValues, lObrigat, bInit, lKey, lNoUpd, lVirtual, cValid)

Onde:

cTitulo Título do campo;

cTooltip Tooltip do campo;

cIdField Id do Field;

cTipo Tipo do campo;

nTamanho Tamanho do campo;

nDecimal Decimal do campo;

bValid Code-block de validação do campo;

bWhen Code-block de validação do modo de edição do campo;

aValues Lista de valores permitido do campo;

lObrigat Indica se o campo tem preenchimento obrigatório;

bInit Code-block de inicialização do campo;

lKey Indica se trata de um campo chave;

lNoUpd Indica se o campo não pode receber valor em uma operação de *update***;**

lVirtual Indica se o campo é virtual;

A seguir exemplificamos o seu uso:

```
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
```

Sua sintaxe para a *interface (View)* é:

AddField(cIdField, cOrdem, cTitulo, cDescric, aHelp, cType, cPicture, bPictVar, cLookUp, lCanChange, cFolder, cGroup, aComboValues, nMaxLenCombo, cIniBrow, lVirtual, cPictVar, lInsertLine)

Onde:

cIdField	Nome do Campo;
cOrdem	Ordem;
cTitulo	Título do campo;
cDescric	Descrição completa do campo;
aHelp	Array com Help;
cType	Tipo do campo;
cPicture	Picture;
bPictVar	Bloco de PictureVar;
cLookUp	Consulta F3;
lCanChange	Indica se o campo é editável;

cFolder	Pasta do campo;
cGroup	Agrupamento do campo;
aComboValues	Lista de valores permitido do campo (combo);
nMaxLenCombo	Tamanho máximo da maior opção do combo;
cIniBrow	Inicializador de *Browse;*
lVirtual	Indica se o campo é virtual;
cPictVar	Picture Variável;

Abaixo exemplificamos os seu uso::

```
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   Titulo 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
```

Obs.: Os campos do tipo lógico serão exibidos como um *checkbox* na *interface (View)*

10.5 Formatação de bloco de código para a estrutura (FWBuildFeature)

Algumas propriedades dos campos da estrutura pedem uma construção específica de bloco de código. Ao se atribuir ou manipular essas propriedades elas devem ser informadas no padrão que o *MVC* espera.

Ao se tratar essas propriedade para o uso em aplicações deve-se usar a função ***FWBuildFeature*** para construí-la.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Exemplo:

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

Onde o 1º parâmetro indica qual a propriedade a ser construída e o 2º o conteúdo a ser atribuído. O 2º parâmetro deve sempre ser ou retornar um dado do tipo caracter.

As propriedades que precisam ser tratadas com esta função são:

STRUCT_FEATURE_VALID Para a **validação**

STRUCT_FEATURE_WHEN Para o **modo de edição**

STRUCT_FEATURE_INIPAD Para o **inicializador padrão**

STRUCT_FEATURE_PICTVAR Para **PictureVar**

Os nomes de propriedades citados acima são na verdade ***#DEFINE***. Para utilizar este ***#DEFINE*** é preciso incluir a seguinte diretiva no fonte:

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

Obs.: Utlize sempre esta a função ***FWBuildFeature*** para a construção das propriedades do contrário poderão ocorrer erros na aplicação, tal como a não atualização das variáveis de memória para os componentes de formulário.

10.6 Campos do tipo MEMO virtuais (FWMemoVirtual)

Alguns campos do tipo MEMO utilizam-se de tabelas para a gravação de seus valores (SYP[3]), esses campos devem ser informados na estrutura para que o *MVC* consiga fazer seu tratamento corretamente.

Usamos para isso a função ***FWMemoVirtual.***

Exemplo:

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

Para estes campos MEMO sempre deve haver outro campo que conterá o código com que o campo MEMO foi armazenado na tabela auxiliar

No exemplo, ***oStruZA1*** é a estrutura que contém os campos MEMO e o segundo parâmetro um vetor bi-dimensional onde cada par relaciona o campo da estrutura que contém o código do campo MEMO com o campo MEMO propriamente dito.

[3] SYP – Tabela do Microsiga Protheus que armazena os dados dos campos do tipo MEMO virtuais

Se a tabela auxiliar a ser utilizada não for a SYP, um 3 parâmetro deverá ser passado no vetor bi-dimensional, como o alias da tabela auxiliar.

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

Observação: Tanto o campo MEMO quanto o campo que armazenará seu código devem fazer parte da estrutura.

10.7 Criação manual de gatilho (AddTrigger / FwStruTrigger)

Se quisermos adicionar um gatilho a uma estrutura já existente, utilizamos o método ***AddTrigger***

Sua sintaxe é:

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

Onde:

cIdField Nome *(ID)* do campo de origem;

cTargetIdField Nome *(ID)* do campo de destino;

bPre Bloco de código de validação da execução do gatilho;

bSetValue Bloco de código de execução do gatilho;

Os blocos de código deste método pendem uma construção específica. Ao se atribuir ou manipular essas propriedades devem ser informadas no padrão que o *MVC* espera.

Para facilitar a construção do gatilho foi criada a função ***FwStruTrigger***, ela retorna um *array* com 4 elementos já formatados para uso no ***AddTrigger.***

Sua sintaxe é:

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

Onde:

cDom Campo Domínio;

cCDom Campo de Contradomínio;

cRegra Regra de Preenchimento;

lSeek Se posicionara ou não antes da execução do gatilhos;

cAlias Alias da tabela a ser posicionada;

nOrdem Ordem da tabela a ser posicionada;

cChave Chave de busca da tabela a ser posicionada;

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

cCondic Condição para execução do gatilho ;

Exemplificando:

```
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 Retirando as pastas de uma estrutura (SetNoFolder)

Se quisermos retirar as pastas que estão configuradas em uma estrutura, por exemplo, pelo uso da função ***FWFormStruct***, usamos o método ***SetNoFolde***r. Da seguinte forma:

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

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

10. 9 Retirando os agrupamentos de campos de uma estrutura (SetNoGroups)

Se quisermos retirar os agrupamentos de campos que estão configuradas em uma estrutura, por exemplo, quando usamos a função ***FWFormStruct***, usamos o método ***SetNoGroups***. Da seguinte forma:

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

11.Criação de campos de total ou contadores (*AddCalc*)

Em *MVC* é possível criar automaticamente um novo componente composto de campos totalizadores ou contadores, um componente de cálculos.

Os campos do componente de cálculos são baseados em componentes de *grid* do modelo. Atualizando o componente de *grid* automaticamente os campos do componente de cálculos serão atualizados.

O ***Addcalc*** é o componente de modelo de dados *(Model)* responsável por isso

Sua sintaxe é:

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

Onde:

cId Identificador do componente de cálculos;

cOwner Identificador do componente superior (*owner*). Não necessariamente é o componente de *grid* de onde virão os dados. Normalmente o superior é ***AddField*** principal do modelo de dados *(Model);*

cIdForm Código do componente de *grid* que contém o campo, a que se refere o campo calculado;

cIdField Nome do campo do componente de *grid* a que se refere o campo calculado;

cIdCalc Identificador (nome) para o campo calculado;

cOperation Identificador da operação a ser realizada.

As operações podem ser:

SUM Faz a soma do campo do componente de *grid;*

COUNT Faz a contagem do campo do componente de *grid;*

AVG Faz a média do campo do componente de *grid;*

FORMULA Executa uma fórmula para o campo do componente de *grid;*

bCond Condição para avaliação do campo calculado. Recebe como parâmetro o objeto do modelo. Retornando .T. (verdadeiro) executa a operação do campo calculado;

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

bInitValue Bloco de código para o valor inicial para o campo calculado. Recebe como parâmetro o objeto do modelo;

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

cTitle Título para o campo calculado;

bFormula Fórmula a ser utilizada quando o parâmetro cOperation é do tipo FORMULA.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Recebe como parâmetros: o objeto do modelo, o valor da atual do campo fórmula, o conteúdo do campo do componente de *grid*, campo lógico indicando se é uma execução de soma (.T. (verdadeiro)) ou subtração (.F. (falso));

O valor retornado será atribuído ao campo calculado;

Exemplo:

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

nTamanho Tamanho do campo calculado (Se não for informado usa o tamanho padrão). Os tamanhos padrões para as operações são:

SUM Será o tamanho do campo do componente de *grid* + 3;

Se o campo do componente de *grid* tiver o tamanho de 9, o campo calculado terá 12.

COUNT Será o tamanho será fixo em 6;

AVG Será o tamanho do campo do componente de *grid*. Se o campo do componente de *grid* tiver o tamanho de 9, o campo calculado terá 9;

FORMULA Será o tamanho do campo do componente de *grid* + 3. Se o campo do componente de *grid* tiver o tamanho de 9, o campo calculado terá 12;

nDecimal Número de casas decimais do campo calculado;

Observação: Para as operações de **SUM** e **AVG** o campo do componente de *grid* tem de ser do tipo numérico.

Exemplo:

```
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' )
...
```

Onde:

COMP022CALC1 É o identificador do componente de cálculos;

ZA1MASTER É o identificador do componente superior (owner);

ZA2DETAIL É o código do componente de grid de onde virão os dados;

ZA2_AUTOR É o nome do campo do componente de grid a que se refere o campo calculado;

ZA2__TOT01 É o Identificador (nome) para o campo calculado;

COUNT É o Identificador da operação a ser realizada;

{ \| oFW \| COMP022CAL(oFW, .T.) } É a condição para avaliação do campo calculado;

'Total Pares' É o título para o campo calculado;

Na ***ViewDef*** também temos que fazer a definição do componente de cálculo. Os dados usados em um componente de cálculo são baseados em um componente de *grid*, porém, a sua exibição se dá da mesma forma que um componente de formulário, por utilizarmos para o componente de cálculo o ***AddField*** e para obtermos a estrutura que foi criada na ***ModelDef*** usamos ***FWCalcStruct***.

Exemplo:

```
Static Funcion 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' )
...
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

12.Outras funções para *MVC*

Algumas funções podem ser especialmente úteis durante o desenvolvimento de uma aplicação.

12.1 Execução direta da interface (FWExecView)

Faz a execução da *interface (View)* com uma determinada operação.

Esta função instancia a *interface (View)* e conseqüentemente o modelo de dados *(Model)* com as operações de **visualizar, incluir, alterar ou excluir**. A intenção dela é fazer similarmente o que faziam as funções ***AXVISUAL***, ***AXINCLI***, ***AXALTERA*** e ***AXDELETA***.

Sua sintaxe é:

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

Onde:

cTitulo Título da janela;

cPrograma Nome do programa-fonte;

nOperation Indica o código de operação (inclusão, alteração ou exclusão);

oDlg Objeto da janela em que o View deve ser colocado. Se não informado, uma nova janela será criada;

bCloseOnOK Indica se a janela deve ser fechada ao final da operação. Se ele retornar .T. (verdadeiro) fecha a janela;

bOk Bloco executado no acionamento do botão confirmar que retornando .F. (falso) impedirá o fechamento da janela;

nPercReducao Se informado reduz a janela em percentualmente;

aEnableButtons Indica os botões da barra de botões que estarão habilitados;

bCancel Bloco executado no acionamento do botão cancelar que retornando .F. (falso) impedirá o fechamento da janela;

O retorno desta função será:

0 Se o usuário finalizar a operação com o botão confirmar;

1 Se o usuário finalizar a operação com o botão cancelar;

Exemplo:

```
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 Modelo de dados ativo (FWModelActive)

Em uma aplicação podemos trabalhar com mais de um modelo de dados *(Model)*. A função ***FWModelActive*** retorna o modelo de dados *(Model)* que está ativo no momento.

Exemplo:

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

Para definir qual o modelo de dados *(Model)* ativo:

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

12.3 Interface ativa (FWViewActive)

Em uma aplicação podemos trabalhar com mais de uma *interface (View)*. A função ***FWViewActive*** retorna a *interface (View)* que está ativa no momento.

Exemplo:

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

Para definir qual a *interface (View)*:

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

12.4 Carregar o modelo de dados de uma aplicação já existente (FWLoadModel)

Para criarmos um objeto com o modelo de dados de uma aplicação, utilizamos o função ***FWLoadModel.***

Sua sintaxe é:

FWLoadModel(<nome do fonte>)

Exemplo:

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

12.5 Carregar a interface de uma aplicação já existente (FWLoadView)

Para criarmos um objeto com o modelo de dados de uma aplicação, utilizamos o função ***FWLoadView.***

Sua sintaxe é:

FWLoadView (<nome do fonte>)

Exemplo:

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

12.6 Carregar a menu de uma aplicação já existente (FWLoadMenudef)

Para criarmos um vetor com as opções de menu de uma aplicação, utilizamos a função ***FWLoadMenudef.***

Sua sintaxe é:

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

Exemplo:

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

12.7 Obtenção de menu padrão (FWMVCMenu)

Podemos criar um menu com opções padrão para o MVC utilizando a função ***FWMVCMENU.***

Sua sintaxe é:

FWMVCMENU (<nome do fonte>)

Exemplo:

```
//---------------------------------------

Static Function MenuDef()

Return FWMVCMenu( 'COMP011_MVC' ))
```

Será criado um menu padrão com as opções: **Visualizar, Incluir, Alterar, Excluir, Imprimir e Copiar**.

13.Browse com coluna de marcação (FWMarkBrowse)

Se quisermos construir uma aplicação com um *Browse* que utilize uma coluna para marcação, similarmente a função ***MarkBrowse*** no *AdvPL* tradicional***,*** utilizaremos a classe ***FWMarkBrowse***.

Assim como a ***FWmBrowse*** (ver cap. 0 3.Aplicações com Browses (*FWMBrowse*), a ***FWMarkBrowse*** não é exclusivamente do *MVC* pode ser utilizada também por aplicações que não o utilizam.

Neste conteúdo não nos aprofundaremos nos recursos da ***FWMarkBrowse***, falaremos aqui de suas principais funções e características para uso em aplicações com *MVC*.

Como premissa, é preciso que haja um campo na tabela do tipo caracter com o tamanho de 2 e que receberá fisicamente a marca. Será gerada uma marca diferente cada vez que a ***FWMarkBrowse*** for executada.

Iniciaremos a construção básica de um ***FWMarkBrowse.***

Primeiramente deve-se criar um objeto ***Browse*** da seguinte forma:

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

Definimos a tabela que será exibida na ***Browse*** pelo método ***SetAlias***. As colunas, ordens, etc. para a exibição serão obtidos através do metadados (dicionários)

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

Definimos o título que será exibido como método ***SetDescription.***

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

Definimos qual será o campo da tabela que receberá a marca física.

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

E ao final ativamos a classe

```
oMark:Activate()
```

Com esta estrutura básica construímos uma aplicação com ***Browse***.

Mas por enquanto temos apenas o ***Browse*** com coluna de marcação, precisamos definir uma

ação para os itens marcados. Para isso, podemos colocar no ***MenuDef*** da aplicação a função que tratará os marcados.

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

Na função que tratará os marcados será preciso identificar se um registro esta ou não marcado. Para sabermos a marca que esta sendo utilizado no momento usamos o método ***Mark.***

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

E para saber se o registro está marcado usamos o método ***IsMark*** passando como parâmetro a marca.

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

É possível também colocar outras opções como visualizar ou alterar no menu de opções (***MenuDef***), mas será preciso criar também o modelo de dados *(Model)* e a *interface (View).*

Todas as outras características da ***FWMBrowse*** também se aplicam a ***FWMarkBrowse*** como legendas, filtros, detalhes, etc.

Um recurso que a ***FWMarkBrowse*** tem é o controle de marcação exclusiva do registro pelo usuário.

Onde se 2 usuários abrirem o mesmo ***FWMarkBrowse*** e tentarem marcar o mesmo registro o próprio ***FWMarkBrowse*** permitirá que apenas um deles execute a marcação. Para habilitar esta característica usamos o método ***SetSemaphore***.

Abaixo, segue um exemplo da utilização do ***FWMarkBrowse***

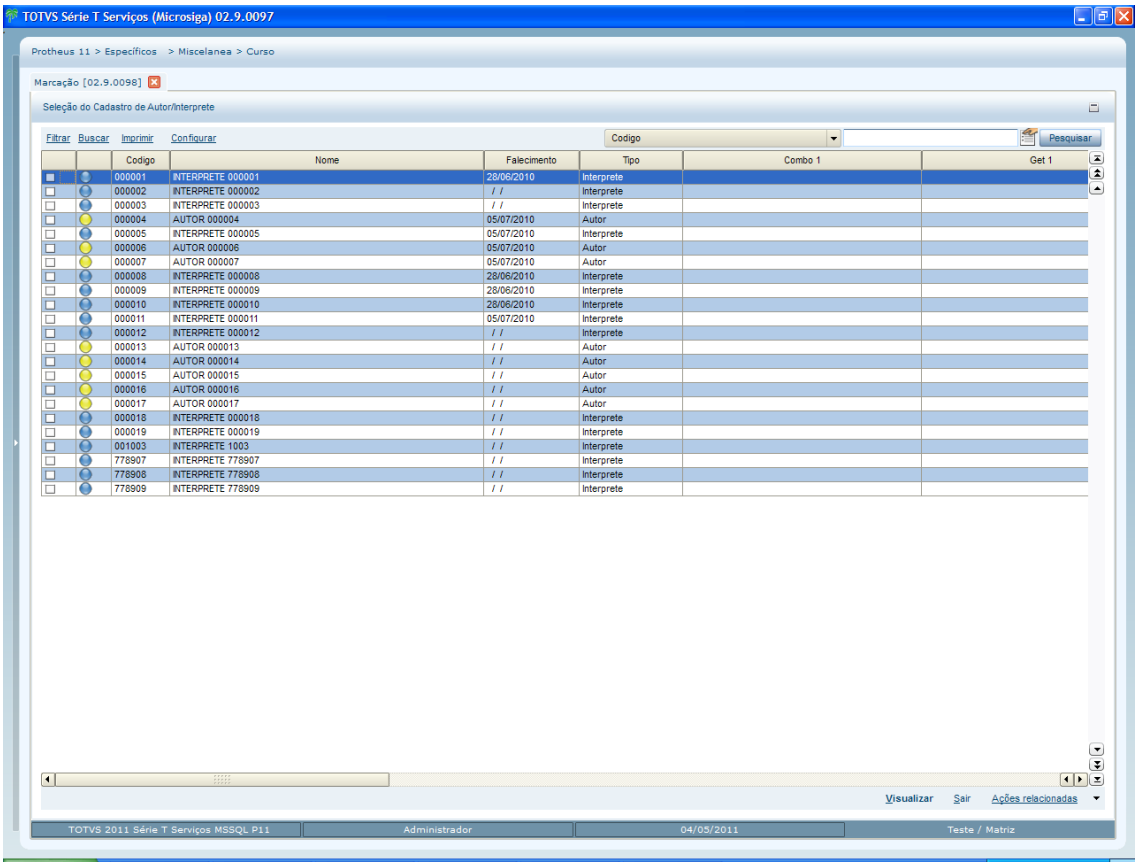
```
User Function COMP025_MVC()
Private oMark

// 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
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
        ZA0->( dbSkip() )
End

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

Return NIL
```

Visualmente teremos:



14.Múltiplos Browses

Com a utilização da classe ***FWmBrowse*** podemos escrever aplicações com mais de um objeto desta classe, ou seja, podemos escrever aplicações que trabalharão com múltiplas ***Browses***.

Podemos por exemplo desenvolver uma aplicação para os pedidos de venda, onde teremos uma ***Browse*** com os cabeçalhos dos itens e outra com os itens na mesma tela e conforme formos navegando pelos registros da ***Browse*** de cabeçalho, automaticamente o itens são atualizados na outra ***Browse.***

Para isso, basta criarmos na nossa aplicação 2 objetos da ***FWMBrowse*** e relacioná-los entre si. Abaixo descreveremos como fazer isso. Criaremos uma aplicação com 3 ***Browses.***

Primeiro criamos uma tela ***Dialog*** comum, cada um dos ***Browses*** deve estar **ancorado** em um objeto ***container***, para isso usaremos o ***FWLayer*** com 2 linhas e em uma dessas linhas colocaremos 2 colunas.

Para mais detalhes do ***FWLayer*** consulte a documentação específica no TDN[4].

```
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 conteiner 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
```

[4] TDN – TOTVS Developer Network portal voltado a desenvolvedores do Microsiga Protheus

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
//
// 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
```

Feito isso criamos as 3 **Browses** conforme o descrito no capítulo **0 3.Aplicações com *Browses* (*FWMBrowse*).**

Este é o **1º *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()
```

Note 2 métodos definidos neste ***Browse: SetProfileID*** e ***ForceQuitButton***

O método ***SetProfileID*** define um identificador *(ID)* para o ***Browse***, isso é necessário já que teremos mais de um ***Browse*** na aplicação.

O método ***ForceQuitButton*** faz com que o botão **Sair** seja exibido nas opções deste ***Browse***. Como haverá mais de um ***Browse*** o botão **Sair** não será colocado automaticamente em nenhum deles, este método faz com que ele apareça no ***Browse.***

Note também que utilizamos o método ***SetMenuDef*** para definir de qual fonte deverá ser utilizado para a obter o ***MenuDef***. Quando não utilizamos o ***SetMenuDef*** automaticamente o ***Browse*** busca no próprio fonte onde ele se encontra o ***Menudef*** a ser usado.

Estes são **o 2º e 3º *Browses:***

```
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()
```

Note que nestes ***Browses*** utilizamos o método ***SetMenuDef*** com uma referência vazia, como queremos que apenas o ***Browse*** principal tenha botões de ação, se não definimos o ***SetMenuDef***, automaticamente, o ***Browse*** busca no próprio fonte onde ele se encontra e com a referência vazia não são exibidos botões.

Agora que definimos os ***Browses*** precisamos fazer o relacionamento entre eles, para que ao efetuar o movimento em um, automaticamente os outros sejam atualizados.

Para criar o relacionamento utilizaremos a classe ***FWBrwRelation***. Similarmente ao que relacionamento entre entidades feito no modelo de dados *(Model)* é preciso dizer quais as chaves de relacionamento do **filho** para **o pai**.

Instanciaremos o ***FWBrwRelation*** e usaremos seu método ***AddRelation***.

A sintaxe deste método do ***FWBrwRelation*** é:

AddRelation(<o*Browse*Pai>, <o*Browse*Filho>, <Vetor com os campos de relacionamento>)

Como temos 3 *Browse*s teremos 2 relacionamentos:

```
oRelacZA4:= FWBrwRelation():New()
oRelacZA4:AddRelation( oBrowseUp  , oBrowseLeft , { { 'ZA4_FILIAL', 'xFilial( "ZA4" )' }, {
'ZA4_ALBUM' , 'ZA3_ALBUM'  } } )
oRelacZA4:Activate()
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
oRelacZA5:= FWBrwRelation():New()
oRelacZA5:AddRelation( oBrowseLeft, oBrowseRight, { { 'ZA5_FILIAL', 'xFilial( "ZA5" )' }, { 'ZA5_ALBUM' , 'ZA4_ALBUM'  }, { 'ZA5_MUSICA', 'ZA4_MUSICA' } } )
oRelacZA5:Activate()
```

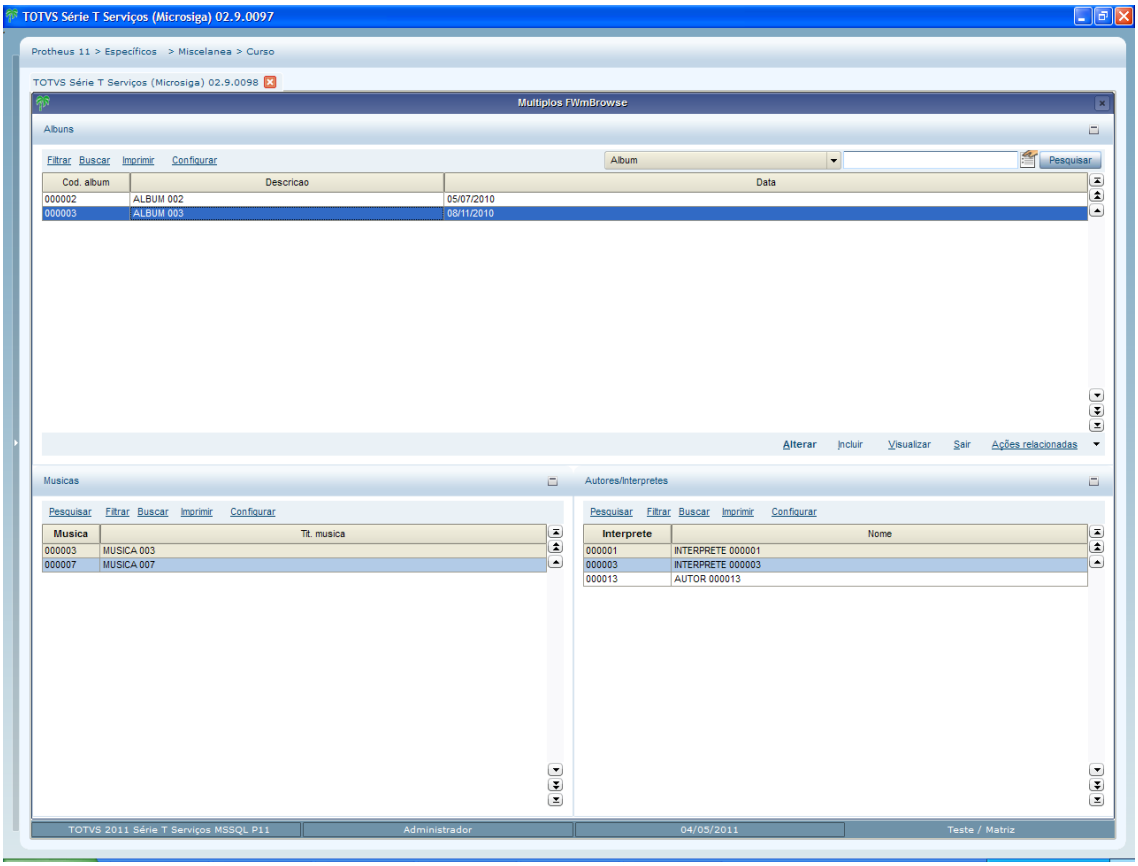
Segue um exemplo completo da aplicação com múltiplas **Browses**:

```
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 conteiner 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' )  // Pego o objeto do pedaço esquerdo
oPanelRight := oFWLayer:GetColPanel( 'RIGHT', 'DOWN' )  // Pego 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' )
```

Visualmente teremos:



Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

15.Rotina automática

Quando uma aplicação é desenvolvida utilizando-se o conceito de *MVC* e suas classes, pode-se fazer uso de seu modelo de dados em outras aplicações, similarmente ao que seria uma **rotina automática** no desenvolvimento tradicional.

Não existe mais o uso da função ***MSExecAuto***. A idéia básica é instanciar o modelo de dados *(Model)* que se deseja, atribuir os valores a ele e fazer a validação.

Para melhor entendimento, usaremos de exemplo o fonte abaixo, onde se faz em *MVC* o que seria uma **rotina automática** para importação de um cadastro simples.

Observe os comentários.

```
//-------------------------------------------------------------------
// 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
             ApMsgInfo( '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 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 instânciamento 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 atribuirmos 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
		// È 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]  ) + ']' )
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
	MostraErro()
EndIf

// Desativamos o Model
oModel:DeActivate()

Return lRet
```

Neste outro exemplo, temos a importação para um modelo de dados onde há uma estrutura de ***Master-Detail*** (**Pai-Filho**). Também o que faremos é instanciar o modelo de dados *(Model)* que desejamos, atribuir os valores à ele e fazer a validação, só que faremos isso para as duas entidades.

Observe os comentários:

```
//-------------------------------------------------------------------
// 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
		ApMsgInfo( '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 instânciamento 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 atribuirmos 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

                  // È feita a atribuição do dado aos campo do Model do cabeçalho

                  If !( lAux := oModel:SetValue( cMaster + 'MASTER', aCpoMaster[nI][1],
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
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()
nItErro  := 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
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
        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
```

Uma situação que poderá ser encontrada, nos casos em que se está convertendo uma aplicação já existente para a estrutura de *MVC,* é o fato da aplicação já estar preparada para trabalhar com rotina automática e conseqüentemente pode haver várias outras aplicações que já utilizam essa rotina automática.

A função ***FWMVCRotAuto*** foi criada para que não seja necessário que estas aplicações, que hoje usam a chamada da rotina padrão, mudem sua forma de trabalhar, já a aplicação foi convertida para *MVC*.

A função utiliza os parâmetros passados no formato anterior de rotina automática ***(MSEXECAUTO)*** e faz o instanciamento do model, atribuição de valores e validação no formato *MVC*, garantindo os programas legados.

Sua sintaxe é:

```
FWMVCRotAuto( oModel, cAlias, nOpcAuto, aAuto, lSeek, lPos )
```

Onde:

oModel Objeto com o modelo do formulário de dados;

cAlias Alias do *Browse* principal;

nOpcAuto Código de identificação do tipo de processamento da rotina automática;

[3] Inclusão

[4] Alteração

[5] Exclusão

aAuto Array com os dados da rotina automática na seguinte estrutura;

[n][1] Código do formulário do Modelo que terá uma atribuição;

[n][2] Array padrão dos dados da EnchAuto e GetDAuto, conforme documentação anterior;

lSeek Indica se o arquivo principal deve ser posicionado com base nos dados fornecidos;

lPos Indica se o nOpc não deve ser calculado com base no aRotina;

Assim a aplicação em *MVC* que foi convertida poderá trabalhar das duas formas:

- A rotina automática e o
- Instânciamento do model.

No exemplo a seguir temos uma rotina de cadastro onde há o tratamento para isso, se os dados ***xRotAuto***, ***nOpcAuto*** forem passados, indica que a aplicação foi chamada por rotina automática e assim usamos a ***FWMVCRotAuto.***

E essa construção não impede que em outras aplicações também se instancie o modelo de dados *(Model)* diretamente.

```
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
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

16.Pontos de entrada no *MVC*

Pontos de entrada são desvios controlados executados no decorrer das aplicações.

Ao se escrever uma aplicação utilizando o *MVC*, automaticamente já estarão disponíveis pontos de entrada pré-definidos.

A idéia de ponto de entrada, para fontes desenvolvidos utilizando-se o conceito de *MVC* e suas classes, é um pouco diferente das aplicações desenvolvidas de maneira convencional.

Nos fontes convencionais temos um **nome** para cada ponto de entrada criado, por exemplo, na rotina ***MATA010*** – Cadastro de Produtos, temos os pontos de entrada: ***MT010BRW***, ***MTA010OK***, ***MT010CAN***, etc. Em *MVC*, não é desta forma.

Em *MVC* criamos um único ponto de entrada e este é chamado em vários momentos dentro da aplicação desenvolvida.

Este ponto de entrada único deve ser uma ***User Function*** e ter como nome o identificador *(ID)* do modelo de dados *(Model)* do fonte. Peguemos de exemplo um fonte do Modulo Jurídico: ***JURA001***. Neste fonte o identificador *(ID)* do modelo de dados (definido na função ***ModelDef***) é também ***JURA001***, portanto ao se escrever o ponto de entrada desta aplicação, faríamos:

```
User Function JURA001()
Local aParam := PARAMIXB
Local xRet   := .T.
...
Return xRet
```

O ponto de entrada criado recebe via parâmetro (***PARAMIXB***) um vetor com informações referentes à aplicação. Estes parâmetros variam para cada situação, em comum todos eles tem os 3 primeiros elementos que são listados abaixo, no quadro seguinte existe a relação de parâmetros para cada ID:

Posições do array de parâmetros comuns a todos os IDs:

POS.	TIPO	DESCRIÇÃO
1	O	Objeto do formulário ou do modelo, conforme o caso
2	C	ID do local de execução do ponto de entrada
3	C	ID do formulário

Como já foi dito, o ponto de entrada é chamado em vários momentos dentro da aplicação, na 2ª posição da estrutura do vetor é passado um identificador *(ID)* que identifica qual é esse momento. Ela pode ter como conteúdo:

ID DO PONTO DE ENTRADA	MOMENTO DE EXECUÇÃO DO PONTO DE ENTRADA
MODELPRE	Antes da alteração de qualquer campo do modelo. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Requer um retorno lógico.
MODELPOS	Na validação total do modelo. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Requer um retorno lógico.
FORMPRE	Antes da alteração de qualquer campo do formulário. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Requer um retorno lógico.
FORMPOS	Na validação total do formulário. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Requer um retorno lógico.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

FORMLINEPRE	Antes da alteração da linha do formulário FWFORM*GRID*. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. 4 N Número da Linha da FWFORM*GRID.* 5 C Ação da FWFORM*GRID.* 6 C Id do campo. Retorno: Requer um retorno lógico.
FORMLINEPOS	Na validação total da linha do formulário FWFORM*GRID*. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. 4 N Número da Linha da FWFORM*GRID.* Retorno: Requer um retorno lógico.
MODELCOMMITTTS	Após a gravação total do modelo e dentro da transação. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Não espera retorno.
MODELCOMMITNTTS	Após a gravação total do modelo e fora da transação. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Não espera retorno.

FORMCOMMITTTSPRE	Antes da gravação da tabela do formulário. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. 4 L Se .T. (verdadeiro) indica novo registro (Inclusão) se .F. (falso) registro já existente (Alteração / Exclusão) . Retorno: Não espera retorno.
FORMCOMMITTTSPOS	Após a gravação da tabela do formulário. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. 4 L Se .T. (verdadeiro) indica novo registro (Inclusão) se .F. (falso) registro já existente (Alteração / Exclusão). Retorno: Não espera retorno.
FORMCANCEL	No cancelamento do botão. Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Requer um retorno lógico.
MODELVLDACTIVE	Na ativação do modelo Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Requer um retorno lógico.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

BUTTONBAR	Para a inclusão de botões na ControlBar. Para criar os botões deve-se retornar um array bi-dimensional com a seguinte estrutura de cada item: 1 C Titulo para o botão. 2 C Nome do Bitmap para exibição. 3 B CodeBlock a ser executado. 4 C ToolTip (Opcional). Parâmetros Recebidos: 1 O Objeto do formulário ou do modelo, conforme o caso. 2 C ID do local de execução do ponto de entrada. 3 C ID do formulário. Retorno: Requer um array de retorno com estrutura pré definida.

Observações:

- Quando o modelo de dados possui vários componentes (por exemplo, de *grid*), a 3ª posição do vetor trará o identificador *(ID)* deste componente;

- Quando o tipo de retorno de um determinado momento de execução não for passado ou for passado com o tipo errado será exibida uma mensagem no console avisando sobre isso. Todos IDs que esperam retorno devem ser tratados no ponto de entrada.

Importante:

- Ao se escrever um fonte em *MVC* que será uma ***User Function***, cuidado ao se atribuir o identificador *(ID)* do modelo de dados *(Model)*, pois ele não poderá ter o mesmo nome do fonte (PRW). Se o fonte tiver o nome ***FONT001***, o identificador *(ID)* do modelo de dados *(Model)* não poderá ser também ***FONT001***, pois não será possível criar outra ***User Function*** com o nome de ***FONT001*** (*ID* do modelo de dados) para os pontos de entrada.

Exemplo:

```
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 para MVC

Ao se desenvolver uma aplicação utilizando *MVC*, já estará disponível um *Web Service* para ser utilizado para o recebimento de dados.

Todas as aplicações em *MVC* utilizarão o mesmo *Web Service*, independentemente de sua estrutura ou de quantas entidades ele possua.

O *Web Service* que está disponível para o *MVC* é o ***FWWSMODEL*** .

A idéia básica é que iremos instanciar o *Web Service*, informar qual a aplicação que será utilizada e informar os dados em um formato XML.

17.1 Web Service para modelos de dados que possuem uma entidade

Veremos como construir uma aplicação que utilize o *Web Service* ***FWWSMODEL*** com um modelo de dados *(Model)* que possui apenas uma entidade.

17.2 Instânciamento do Client de Web Service

O instânciamento se dá seguinte forma:

Instânciamento do *Client* do *Web Service* propriamente dito.

```
oMVCWS := WsFwWsModel():New()
```

Definição da URL do ***FWWSMODEL*** no servidor de *Web Services.*

```
oMVCWS:_URL := http://127.0.0.1:8080/ws/FWWSMODEL.apw
```

Definição da aplicação que será usada.

Definimos aqui o nome do fonte que contém a ***ModelDef*** que queremos utilizar.

```
oMVCWS:cModelId := 'COMP011_MVC'
```

17.3 A estrutura do XML utilizada

Como dito anteriormente os dados serão informados em um XML. A estrutura desse XML segue a seguinte hierarquia básica:

```
<ID do Model>
      <ID de Componente>
            <ID de Campo>
                  Conteúdo...
            </ID de Campo
      </ID de Componente >
</ID do Model>
```

A *tag* ***<ID do Model>*** é o que identificador *(ID)* foi definido no modelo de dados (Model) da aplicação *MVC.*

Exemplo:

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Na aplicação temos definido:

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

No XML a tags **<ID do Model>** serão:

```
<COMP011M>
      ...
</COMP011M>
```

A operação que será realizada **inclusão (3), alteração (4)** ou **exclusão (5)** também dever ser informada nesta *tag,* no atributo ***Operation.***

Assim se quisermos fazer uma operação de inclusão teremos:

```
<COMP011M Operation="3">
```

As *tags* ***<ID de Componente>*** são *IDs* dos componentes de formulários ou componente de gris que foram definidos no modelo de dados *(Model)* da aplicação.

Exemplo:

Se na aplicação temos:

```
oModel:AddFields( 'ZA0MASTER' )
```

No XML as *tags* **<ID de Componente>** serão:

```
<ZA0MASTER>
      ...
</ZA0MASTER>
```

O tipo do componente (de formulário ou de *grid*) também deve ser informados nesta *tag* no atributo ***modeltype***. Informe ***FIELDS*** para componentes de formulários e ***GRID*** para componentes de *grid.*

Teríamos então:

```
<ZA0MASTER modeltype="FIELDS">
```

As *tags* ***<ID de Campo>*** serão os nomes dos campos da estrutura do componente, seja formulário ou *grid.*

Assim se na estrutura tivermos os campos ***ZA0_FILIAL***, ***ZA0_ CODIGO*** e ***ZA0_NOME***, por exemplo, teremos:

```
<ZA0_FILIAL>
      ...
</ZA0_FILIAL>

<ZA0_CODIGO>
      ...
</ZA0_CODIGO>
```

```
<ZA0_NOME>
      ...
</ZA0_NOME>
```

A ordem dos campos também deve ser informada nestas *tags,* no atributo ***order.***

```
<ZA0_FILIAL order="1">
      ...
</ZA0_FILIAL>
<ZA0_CODIGO order="2">
      ...
</ZA0_CODIGO >
<ZA0_NOME   order="3">
      ...
</ZA0_NOME>
```

Quando o componente é um formulário (***FIELDS***), os dados propriamente ditos devem ser informados em uma *tag* ***value.***

```
<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 Obtendo a estrutura XML de um modelo de dados

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

(GetXMLData)

Podemos obter a estrutura XML que uma aplicação em *MVC* espera, para isso utilizando o método ***GetXMLData*** do *Web Service*.

Exemplo:

```
oMVCWS:GetXMLData()
```

O XML esperado será informado no atributo ***cGetXMLDataResult*** do WS.

```
cXMLEstrut := oMVCWS:cGetXMLDataResult
```

Utilizando ainda o exemplo acima, teríamos:

```
<?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 Informando os dados XML ao Web Service

O XML contendo os dados dever ser atribuído ao atributo ***cXML*** do objeto de *Web Service.*

Exemplo:

```
oMVCWS:cXML := cXML // variável que contem o XML com os dados
```

17.6 Validando os dados (VldXMLData)

Para submetermos esses dados ao modelo de dados *(Model)* para que sejam validados utilizamos o método ***VldXMLData.***

```
If !oMVCWS:VldXMLData()
      MsgStop( 'Problemas na validação dos dados' + CRLF + WSError() )
EndIf
```

Neste momento os dados são validados, mas não são gravados. O ***VldXMLData*** apenas valida.

Este é um recurso interessante se quisermos fazer uma simulação, por exemplo.

17.7 Validando e gravando os dados (PutXMLData)

A diferença entre o método ***VldXMLData*** e o método ***PutXMLData***, é que o ***PutXMLData*** além de submeter os dados do XML ao modelo de dados para validação, também fará a gravação destes dados se a validação foi bem sucedida.

O resultado é informado no atributo ***lPutXMLDataResult*** e caso haja algum problema, será descrito no atributo ***cVldXMLDataResult*** do objeto de *Web Service.*

```
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 Obtendo o esquema XSD de um modelo de dados (GetSchema)

O XML informado antes da validação das informações pelo modelo de dados *(Model)* será validado pelo *schema* XSD referente ao modelo. Essa validação é feita automaticamente e o XSD é baseado na estrutura do modelo de dados *(Model)*.

Esta validação se refere à estruturação do XML (*tags, níveis, ordens, etc.*) e não aos dados dos XML, a validação dos dados é função da regra de negócios.

Se o desenvolvedor quiser obter o *schema* XSD que será usado, poderá usar o método ***GetSchema.***

Exemplo:

```
If oMVCWS:GetSchema()
	cXMLEsquema := oMVCWS:cGetSchemaResult
EndIf
```

O *schema* XSD é retornado no atributo cGetSchemaResult do objeto de *Web Service*.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

17.9 Exemplo completo de 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()
              cXMLEsquema := 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 para modelos de dados que possuem duas ou mais entidades

Para a construção de *Web Services* que possuam duas ou mais entidades o que será diferente é apenas o XML recebido que terá mais níveis. Observe o fonte exemplo:

```
#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
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
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 estutura 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>'
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
	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.Uso do comando *New Model*

Para facilitar o desenvolvimento foram criados comandos que podem gerar mais simples e rapidamente uma aplicação em *MVC*. É o comando ***New Model.***

Este comando é indicado para aquelas aplicações onde se tem o uso de uma tabela (antiga ***Modelo1***) ou uma tabela não normalizada (cabeçalho e item no mesmo registro), mas com a necessidade de trabalhar em uma estrutura ***master-detail*** (antiga ***Modelo2***) ou onde se tem o uso de duas tabelas em uma estrutura ***master-detail*** (antiga ***Modelo3***).

Utilizando o comando ***New Model*** não é necessário se escrever todas as funções e classes normalmente utilizadas em uma rotina *MVC*. Durante o processo de pré-compilação o ***New Model*** e suas diretivas são transformados em um fonte *MVC* que utiliza ***FWmBrowse***, ***ModelDef***, ***ViewDef*** e eventualmente ***MenuDef***.

A premissa para se utilizar este comando é que se tenha uma das construções citadas acima e as estruturas das tabelas estejam definidas no dicionário SX3. Não poderão ser construídas estruturas manualmente ou se adicionar ou retirar campos das estruturas.

Como este comando é uma diretiva de compilação do tipo ***#COMMAND***, para utilizar este comando é preciso incluir a seguinte diretiva no fonte:

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

À seguir há a sintaxe do comando e exemplos de uso.

18.1 Sintaxe da New Model

Esta é sintaxe do comando ***New Model:***

NEW MODEL

TYPE <nType> ;

DESCRIPTION <cDescription> ;

BROWSE <oBrowse> ;

SOURCE <cSource> ;

MODELID <cModelID> ;

FILTER <cFilter> ;

CANACTIVE *<bSetVldActive> ;*

PRIMARYKEY *<aPrimaryKey> ;*

MASTER *<cMasterAlias> ;*

HEADER *<aHeader,...> ;*

BEFORE *<bBeforeModel> ;*

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

AFTER <*bAfterModel*> ;

COMMIT <*bCommit*> ;

CANCEL <*bCancel*> ;

BEFOREFIELD <*bBeforeField*> ;

AFTERFIELD <*bAfterField*> ;

LOAD <*bFieldLoad*> ;

DETAIL <*cDetailAlias*> ;

BEFORELINE <*bBeforeLine*> ;

AFTERLINE <*bAfterLine*> ;

BEFORE*GRID* <*bBeforeGrid*> ;

AFTER*GRID* <*bAfterGrid*> ;

LOAD*GRID* <*bGridLoad*> ;

RELATION <*aRelation*> ;

ORDERKEY <*cOrder*> ;

UNIQUELINE <*aUniqueLine*> ;

AUTOINCREMENT <*cFieldInc*> ;

OPTIONAL

Onde:

TYPE <nType>

Tipo Numérico - Obrigatório

Tipo de Estrutura

1 = 1 Tabela

2 = 1 Tabela Master/Detail

3 = 2 Tabelas Master/Detail

DESCRIPTION <cDescription>

Tipo Caracter - Obrigatório

Descrição da Rotina

BROWSE <oBrowse>

Tipo Objeto - Obrigatório

Objeto de *Browse* que será utilizado

SOURCE <cSource>

Tipo Caracter - Obrigatório

Nome do Fonte

MODELID <cModelID>

Tipo Caracter - Obrigatório

identificador *(ID)* do Model

FILTER <cFilter>

Tipo Caracter - Opcional

Filtro para *Browse*

CANACTIVE <bSetVldActive>

Tipo Bloco - Opcional

Bloco para validação da ativação do Model. Recebe como parâmetro o Model

Ex. { |oModel| COMP011ACT(oModel) }

PRIMARYKEY <*aPrimaryKey*>

Tipo Array - Opcional

Array com as Chaves primárias do *Browse*, se não for informado ira buscar o X2_UNICO da tabela.

MASTER <cMasterAlias>

Tipo Caracter - Obrigatório

Tabela Principal (Master)

HEADER <*aHeader*>

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Tipo Array - Obrigatório para TYPE = 2

Array com campos que serão considerados no "Cabeçalho"

BEFORE <bBeforeModel>

Tipo Bloco - Opcional

Bloco de Pré Validação do Model. Recebe como parâmetro o Model.

Ex. { |oModel| COMP011PRE(oModel) }

AFTER <bAfterModel>

Tipo Bloco - Opcional

Bloco de Pós Validação do Model Recebe como parâmetro o Model.

Ex. { |oModel| COMP011POS(oModel) }

COMMIT <*bCommit*>

Tipo Bloco - Opcional

Bloco de persistência dos dados (Commit) do Model. Recebe como parâmetro o Model.

Ex. { |oModel| COMP022CM(oModel) }

CANCEL <*bCancel*>

Tipo Bloco - Opcional

Bloco acionado no botão de cancelar. Recebe como parâmetro o Model.

Ex. { |oModel| COMP011CAN(oModel) }

BEFOREFIELD <*bBeforeField*>

Tipo Bloco - Opcional

Bloco de Pré Validação da FORMFIELD da tabela Master. Recebe como parâmetro o ModelField, identificador *(ID)* do local de execução e identificador *(ID)* do Formulário

Ex. { |oMdlF,cId ,cidForm| COMP023FPRE(oMdlF,cId ,cidForm) }

AFTERFIELD <*bAfterField*>

Tipo Bloco - Opcional

Bloco de Pós Validação da FORMFIELD da tabela Master. . Recebe como parâmetro o ModelField, identificador *(ID)* do local de execução e identificador *(ID)* do Formulário

Ex. { |oMdlF,cId ,cidForm| COMP023FPOS(oMdlF,cId ,cidForm) }

LOAD <bFieldLoad>

Tipo Bloco - Opcional

Bloco de Carga dos dados da FORMFIELD da tabela Master

DETAIL <cDetailAlias>

Tipo Caracter - Obrigatório para TYPE = 2 ou 3

Tabela Detail

BEFORELINE <*bBeforeLine*>

Tipo Bloco - Opcional

Bloco de Pré Validação da linha da FORM*GRID* da tabela Detail. Recebe como parâmetro o Model*Grid*, o numero da linha do FORM*GRID*, a ação e o campo da FORM*GRID*.

Ex. { |oMdlG,nLine,cAcao,cCampo| COMP023LPRE(oMdlG, nLine, cAcao, cCampo) }

Usado apenas para TYPE = 2 ou 3

AFTERLINE <*bAfterLine*>

Tipo Bloco - Opcional

Bloco de Pós Validação da linha da FORM*GRID* da tabela Detail. Recebe como parâmetro o Model*Grid* e o numero da linha do FORM*GRID*.

Ex. { |oModelGrid, nLine| COMP022LPOS(oModelGrid, nLine) }

Usado apenas para TYPE = 2 ou 3

BEFOREGRID <bBeforeGrid>

Tipo Bloco - Opcional

Bloco de Pré Validação da FORM*GRID* da tabela Detail. Recebe como parâmetro o Model*Grid*

Usado apenas para TYPE = 2 ou 3

AFTERGRID <bAfterGrid>

Tipo Bloco - Opcional

Bloco de Pré Validação da FORM*GRID* da tabela Detail. Recebe como parâmetro o Model*Grid*

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

Usado apenas para TYPE = 2 ou 3

LOADGRID <bGridLoad>

Tipo Bloco - Opcional

Bloco de Carga dos dados da FORM*GRID* da tabela Detail

Usado apenas para TYPE = 2 ou 3

RELATION <*aRelation*>

Tipo Array - Obrigatório para TYPE = 2 ou 3

Array bidimensional para relacionamento das tabelas Master/Detail

Usado apenas para TYPE = 2 ou 3

ORDERKEY <*cOrder*>

Tipo Caracter - Opcional

Ordenação da FORM*GRID* da tabela Detail

Usado apenas para TYPE = 2 ou 3

UNIQUELINE <*aUniqueLine*>

Tipo Array - Opcional

Array com campos para que poderão ser duplicados na FORM*GRID* da tabela Detail

Usado apenas para TYPE = 2 ou 3

AUTOINCREMENT <*cFieldInc*>

Tipo Array - Opcional

Campos auto incremental para FORM*GRID* da tabela Detail

Usado apenas para TYPE = 2 ou 3

OPTIONAL

Indica se o preenchimento da FORM*GRID* da tabela Detail será opcional

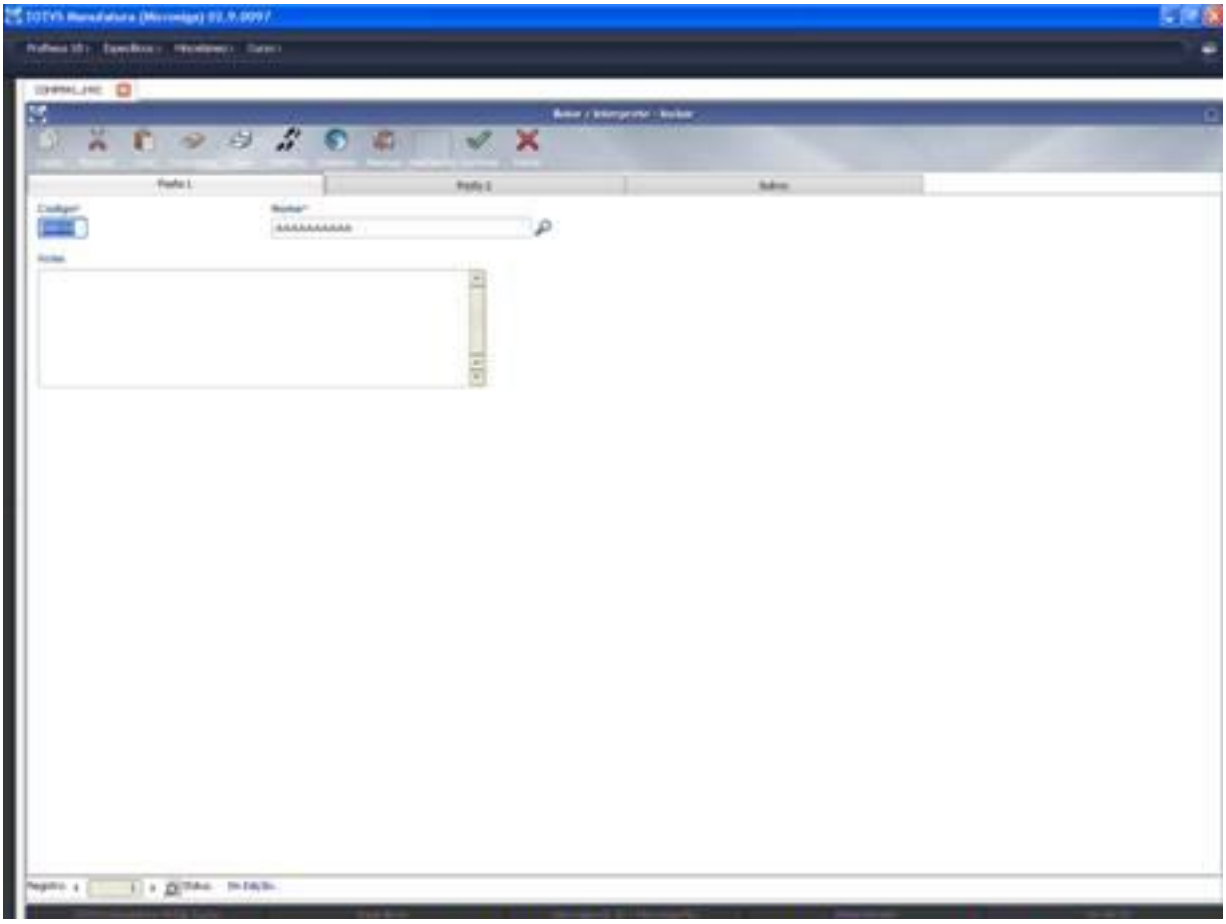
Usado apenas para TYPE = 2 ou 3

Exemplo:

```
//
// 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
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

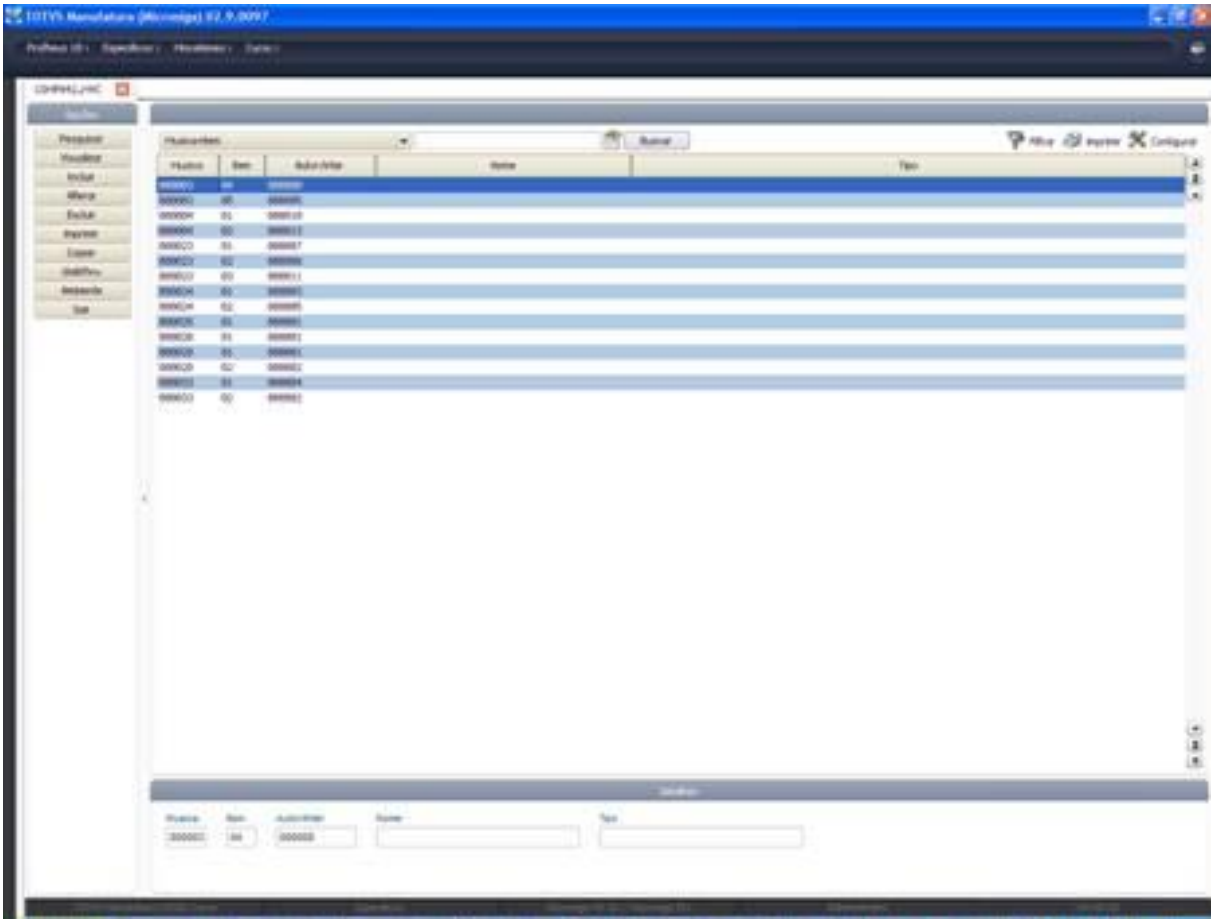
Visualmente teremos:



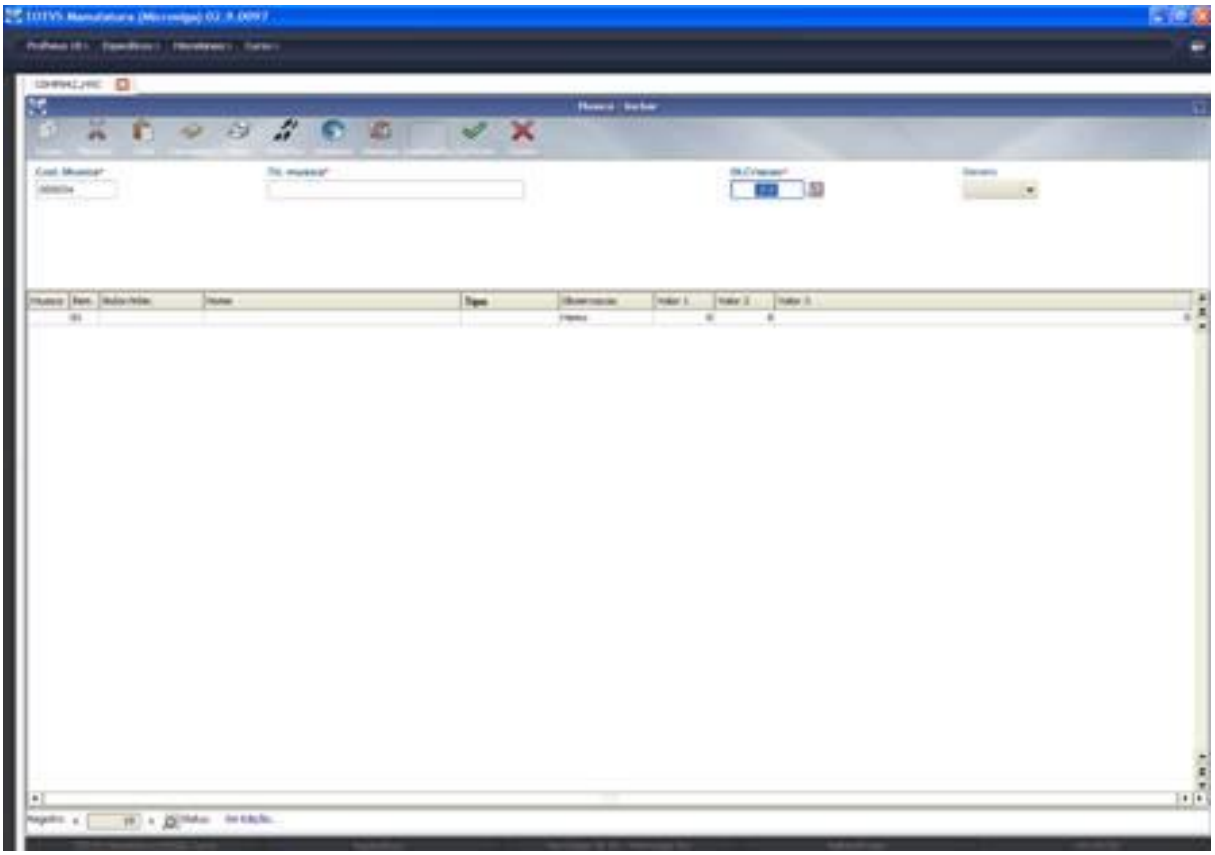
```
//
// Construcao para uma tabela Master/Detail
//
#INCLUDE "PROTHEUS.CH"
#INCLUDE "FWMVCDEF.CH"

User Function COMP042_MVC()
Local oBrowse

NEW MODEL ;
TYPE              2    ;
DESCRIPTION       "Tabela Nao Normalizada" ;
BROWSE            oBrowse          ;
SOURCE            "COMP042_MVC"    ;
MODELID           "MDCOMP042"      ;
MASTER            "ZA2"            ;
HEADER            { 'ZA2_MUSICA', 'ZA2_ITEM' } ;
```

```
RELATION          { { 'ZA2_FILIAL', 'xFilial( "ZA2" )' }, ;
{ 'ZA2_MUSICA', 'ZA2_MUSICA' } } ;
UNIQUELINE        { 'ZA2_AUTOR' } ;
ORDERKEY          ZA2->( IndexKey( 1 ) ) ;
AUTOINCREMENT     'ZA2_ITEM'
Return NIL
```

O Resultado é:



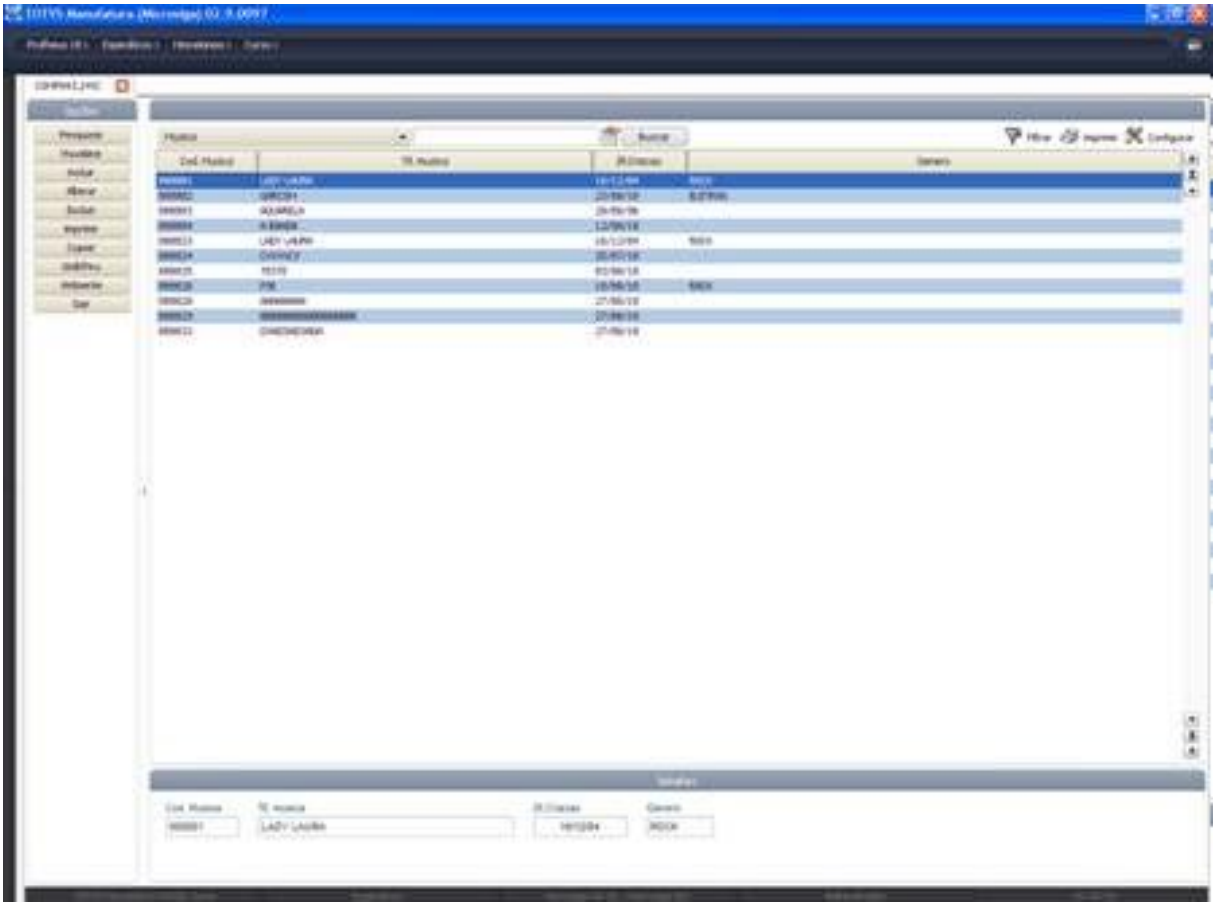
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®



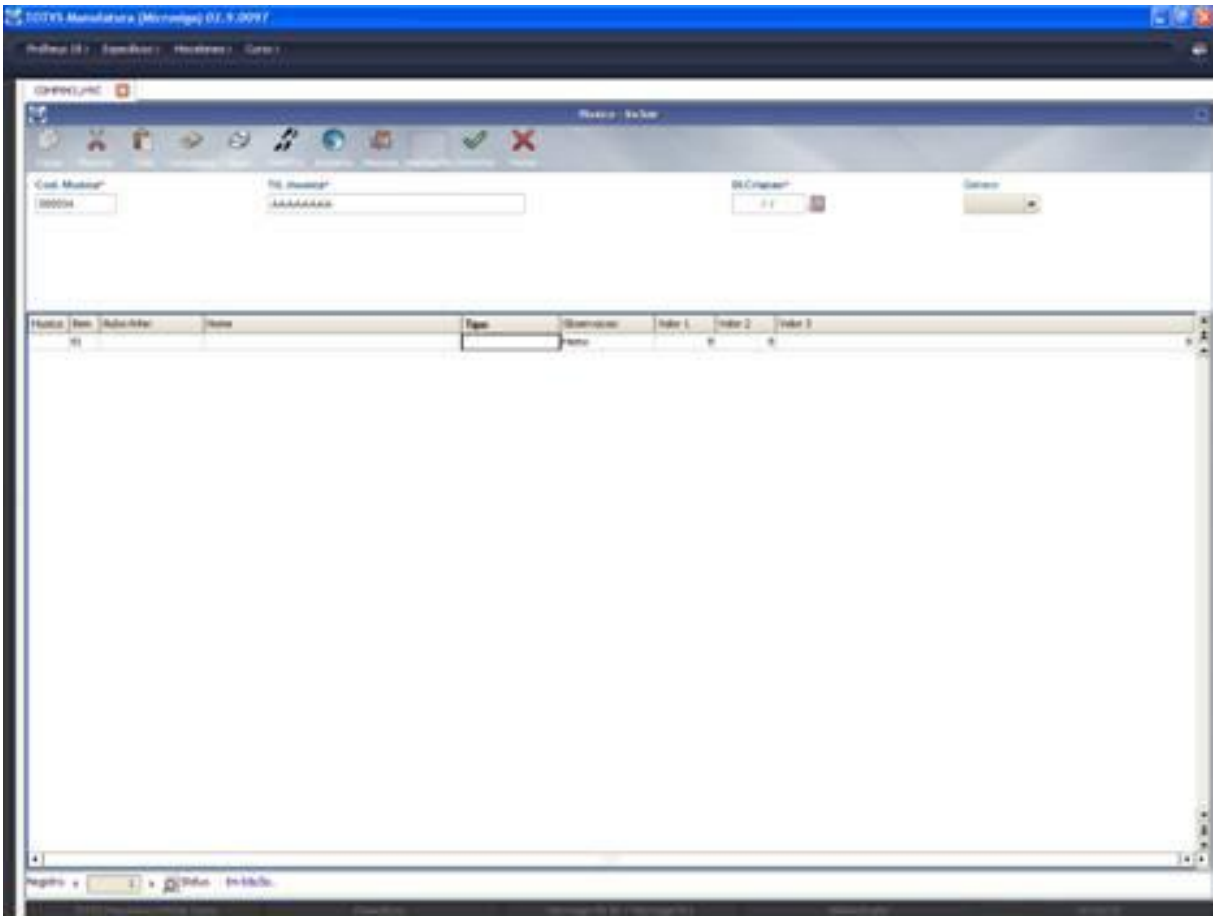
```
//
// Construcao para duas tabelas Master/Detail
//
#INCLUDE "PROTHEUS.CH"
#INCLUDE "FWMVCDEF.CH"

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
```

O Resultado é:



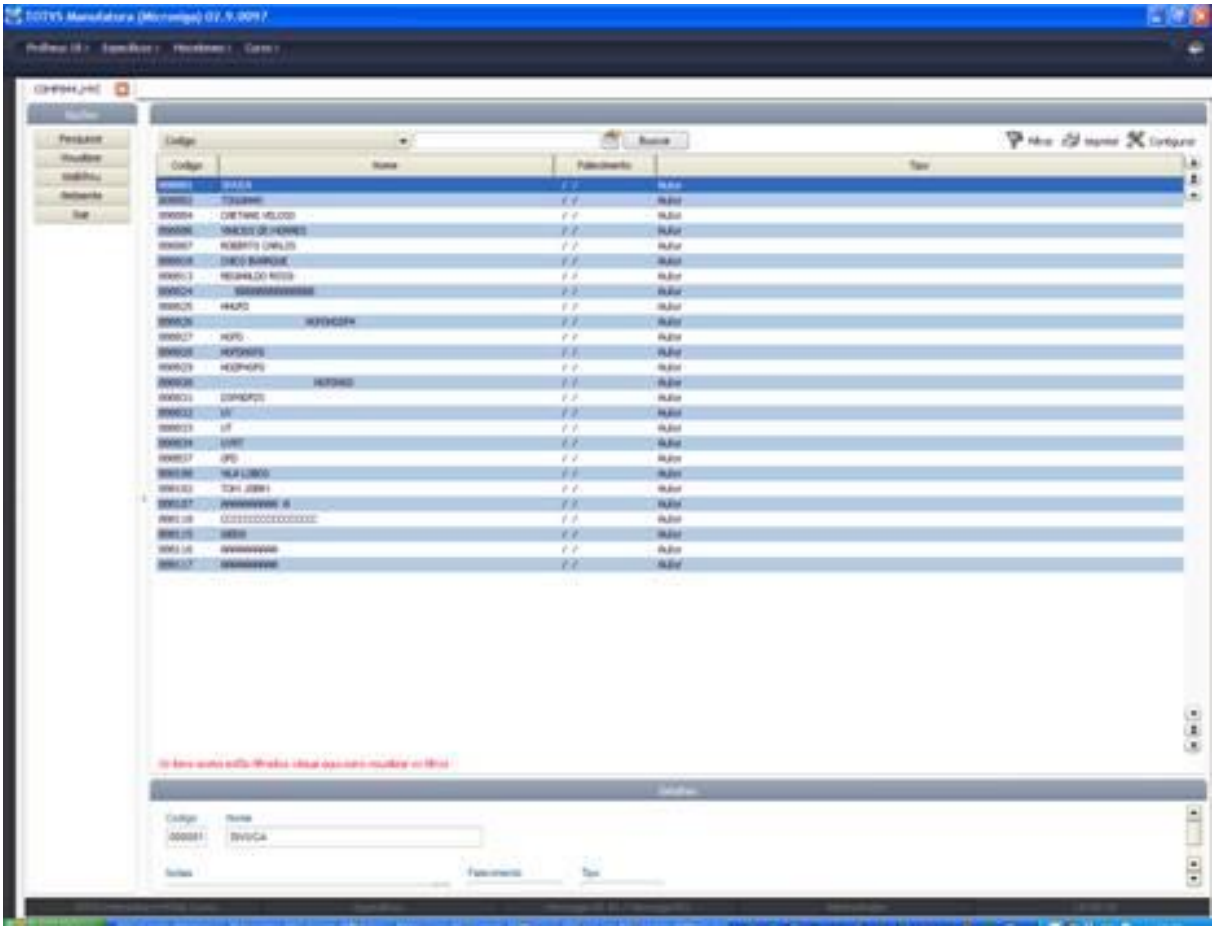
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®



```
//
// 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"   ;
MENUDEF     "COMP044_MVC"   ;
MODELID     "MDCOMP044"     ;
FILTER      "ZA0_TIPO=='1'" ;
MASTER      "ZA0"
```

```
Return NIL
//---------------------------------------------------------------------
Static Function MenuDef()
Local aRotina := {}
ADD OPTION aRotina TITLE 'Pesquisar'  ACTION 'PesqBrw'             OPERATION 1 ACCESS 0
ADD OPTION aRotina TITLE 'Visualizar' ACTION 'VIEWDEF.COMP044_MVC' OPERATION 2 ACCESS 0
Return aRotina
```

O Resultado é:



Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

19.Reutilizando um modelo de dados ou *interface* já existentes

Uma das grandes vantagens na construção de aplicações em *MVC* é a possibilidade de reutilização do modelo de dados (*Model*) ou da *interface* (*View*) em outras aplicações, utilizando o conceito de herança.

Tanto podemos reutilizar os componentes como estão definidos, como também podemos acrescentar novas entidades aos mesmos.

Para fazer isso precisamos dentro da nova aplicação instanciar o modelo de dados (*Model*) ou a *interface* (*View*).

A seguir exemplificamos esse uso.

19.1 Apenas reutilizando os componentes

Neste exemplo realizaremos o modelo de dados (*Model*) e a *interface* (*View*) já existentes em uma aplicação, para a construção de uma nova sem alterações.

Usaremos as funções:

FWLoadModel, ver cap. 0 12.4 Carregar o modelo de dados de uma aplicação já existente (FWLoadModel)) e;

FWLoadView, ver cap. 0 12.5 Carregar *a* interface de uma aplicação já existente (FWLoadView).

Na ***ModelDef*** da nova aplicação instanciamos o modelo de dados (*Model*) da aplicação já existente:

```
Static Function ModelDef()
```

Local oModel := **FWLoadModel**("COMP011_MVC")

```
Return oModel
```

Na ***MenuDef*** da aplicação instanciamos a *interface* (View) de outra aplicação:

```
Static Function ModelDef()
Local oModel := FWViewModel( "COMP011_MVC" )
Return oModel
```

Nos exemplos acima a nova aplicação usará os mesmos componentes da aplicação já existente, no caso, o que está definido na ***ModelDef*** do fonte ***COMP011_MVC.***

Exemplo:

```
#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 Reutilizando e complementando os componentes

Mostraremos agora como reutilizar um componente de *MVC* onde acrescentaremos novas entidades ao mesmo. Só é possível acrescentar novas e não retirar entidades, pois se retirássemos alguma entidade estaríamos quebrando a regra de negócios criada no modelo original.

O ideal para este tipo de uso é criarmos um modelo básico e o incrementarmos conforme a necessidade.

Analisemos primeiro o modelo de dados (*Model*). No exemplo a partir do modelo de dados já existente acrescentaremos uma nova entidade.

Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

O primeiro passo é criar a estrutura da nova entidade, ver cap. **0 5.1 Construção de uma estrutura de dados (FWFormStruct**) para detalhes**.**

```
// Cria a estrutura a ser acrescentada no Modelo de Dados
Local oStruZA6 := FWFormStruct( 1, 'ZA6', /*bAvalCampo*/, /*lViewUsado*/ )
```

Instanciamos o modelo já existente.

```
// Inicia o Model com um Model já existente
Local oModel   := FWLoadModel( 'COMP011_MVC' )
```

No nosso exemplo, acrescentaremos um novo formulário, ver cap. 0 5.3 Criação de um componente de formulários no modelo de dados (AddFields) para detalhes.

Note que em nossa nova aplicação não usamos o ***MPFormModel***, pois estamos apenas acrescentando entidade. O ***MPFormModel*** foi usado na aplicação original.

```
// Adiciona a nova FORMFIELD
oModel:AddFields( 'ZA6MASTER', 'ZA0MASTER', oStruZA6 )
```

Fazemos o relacionamento do novo formulário, ver cap. 0

6.5 Criação de relação entre as entidades do *modelo* (SetRelation).

```
// Faz relacionamento entre os compomentes do model
oModel:SetRelation( 'ZA6MASTER', { { 'ZA6_FILIAL', 'xFilial( "ZA6" )' }, { 'ZA6_CODIGO',
'ZA0_CODIGO' } }, ZA6->( IndexKey( 1 ) ) )
```

Acionamos a descrição do novo formulário.

```
// Adiciona a descricao do novo componente
oModel:GetModel( 'ZA6MASTER' ):SetDescription( 'Complemento dos Dados de Autor/Interprete'
```

E ao final retornamos o novo modelo.

```
Return oModel
```

Com isso criamos um modelo a partir de outro e acrescentamos um novo componente de formulário.

Veremos agora como reutilizar a *interface* (*View*), também acrescentando um novo componente.

O primeiro passo é criar a estrutura da nova entidade ,ver cap. 0 5.1 Construção de uma estrutura de dados (FWFormStruct) .

```
// Cria a estrutura a ser acrescentada na View
Local oStruZA6 := FWFormStruct( 2, 'ZA6' )
```

Instanciaremos o modelo utilizado pela *interface*, note que não instanciaremos o modelo original e sim o modelo da nova aplicação que já tem o novo componente acrescido em seu modelo de dados.

```
// Cria um objeto de Modelo de Dados baseado no ModelDef do fonte informado
Local oModel   := FWLoadModel( 'COMP015_MVC' )
```

Instanciamos a *interface* original

```
// Inicia a View com uma View ja existente
Local oView    := FWLoadView( 'COMP011_MVC' )
```

Adicionamos o novo componente da view e associamos ao criado no modelo, ver cap. 0 5.8 Criação de um componente de *formulários* na interface (AddField) para detalhes.

```
// Adiciona no nosso View um controle do tipo FormFields(antiga enchoice)
oView:AddField( 'VIEW_ZA6', oStruZA6, 'ZA6MASTER' )
```

Temos que criar um ***box*** para o novo componente. É preciso criar sempre um ***box*** vertical dentro de um horizontal e vice-versa como na ***COMP011_MVC*** o ***box*** já existente é horizontal, cria-se um vertical primeiro, para detalhes ver cap. **0 6.13 Exibição dos dados *na* interface (CreateHorizontalBox / CreateVerticalBox).**

```
// 'TELANOVA' é o box existente na interface original
oView:CreateVerticallBox( 'TELANOVA' , 100, 'TELA'     )
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
// Novos Boxes
oView:CreateHorizontalBox( 'SUPERIOR' , 50, 'TELANOVA' )
oView:CreateHorizontalBox( 'INFERIOR' , 50, 'TELANOVA' )
```

Relacionado os componente com os ***box*** de exibição, ver cap. **0 5.10 Relacionando o componente *da* interface (SetOwnerView).**

```
oView:SetOwnerView( 'VIEW_ZA0', 'SUPERIOR' )
oView:SetOwnerView( 'VIEW_ZA6', 'INFERIOR' )
```

E ao final retornamos o novo objeto de *interface*.

```
Return oView
```

Com isso criamos uma *interface* a partir de outra e acrescentamos um novo componente.

Um exemplo de aplicação para este conceito seria a internacionalizacão, onde poderíamos ter um modelo básico e o incrementaríamos conforme a localização.

Para entender melhor a internacionalização, veja o Apêndice A.

Abaixo temos o exemplo completo da aplicação que reutiliza componentes.

19.3 Exemplo completo de uma aplicação que reutiliza componentes de modelo e interface

```
#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 )
```


Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

```
// Faz relacionamento entre os compomentes 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
```

Apêndice A

O Framework *MVC* do Microsiga Protheus e a internacionalização.

Internacionalização (I18N) e localização (L10N) são processos de desenvolvimento e/ou adaptação de softwares, para uma língua e/ou cultura de um país. A internacionalização de um software não fornece um novo Sistema, somente adapta as mensagens do Sistema à língua e à cultura locais. A localização por sua vez, adiciona novos elementos do país ao Sistema, como processos, aspectos legais, entre outros.

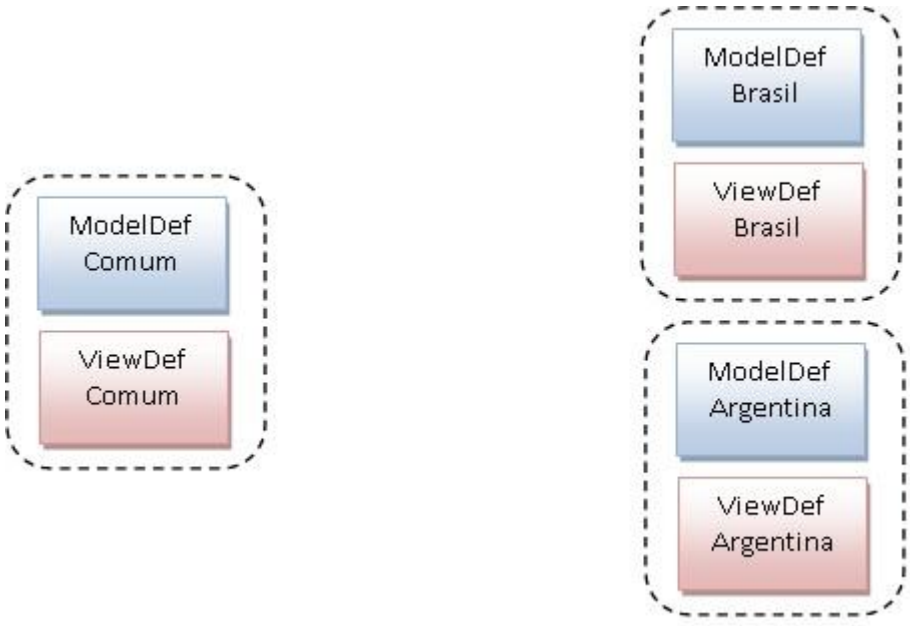
O Framework *MVC* auxilia a localização do Sistema, componentizando o software de forma que a parte comum a todos os países seja desagregada da parte não-comum, incluindo-se *interface* e regra de negócio.

Por exemplo, tome como base o formulário **Nota Fiscal/*Invoice***. Este formulário tem como característica comum em todos os países os elementos: **Origem, Destino, Lista de produtos, Transporte e Faturas.**

Em certos países como o Brasil, é necessário registrar elementos legais, **como impostos, escrituração, códigos de classificação**, entre outros. A alternativa que se tem é duplicar o código ou alterar o código inserindo linhas de código dos elementos localizados. Apesar de esta alternativa funcionar bem no inicio, ao longo do tempo mostra-se impraticável devido ao volume de implementações diferentes para cada país, causando grandes transtornos e um alto custo para a sustentação do Sistema.

O Framework *MVC* traz uma luz racional e simples para este problema. A herança de formulários. É possível construir um formulário comum para a **Nota Fiscal/*Invoice*** que não tenha nenhum elemento de localização e utilizá-lo pela herança, como base para os formulários localizados.

Neste modelo, garante-se a evolução da localização e da parte comum do formulário sem que uma implementação afete a outra reduzindo o custo de sustentação do produto.



A herança do framework MVC pode ocorrer no *Model* e *View* ou somente no *View*.

Neste momento vocês devem estar se perguntado como isto pode ser feito. A resposta esta no par de funções ***FWLoadModel*** e ***FWLoadView***, como poder ser visto no código abaixo:

```
#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)
```

Há inúmeras vantagens neste modelo de desenvolvimento que gostaria de destacar além da componentização, que é o isolamento do código fonte. O isolamento permite que os dois códigos fontes evoluam separadamente, porém pela herança o código localizado sempre irá herdar os benefícios da parte comum, inclusive possibilitando que duas pessoas interajam simultaneamente no mesmo processo sem que um prejudique o trabalho do outro.

Índice Remissivo