MVC

Protheus – Versão 12

Sumário

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

1. Objetivo

Ao final do treinamento deverá ter desenvolvido os seguintes conceitos:

a) Conceitos a serem aprendidos

- Arquitetura *Model-View-Controller* ou MVC
- introdução as técnicas de programação voltado ao MVC
- introdução aos conceitos entre as camadas do MVC

b) Habilidades e técnicas a serem aprendidas

- desenvolvimento de aplicações voltadas ao ERP Protheus
- análise de fontes de média complexidade
- desenvolvimento de fontes com mais de uma entidade

1.1. Introdução

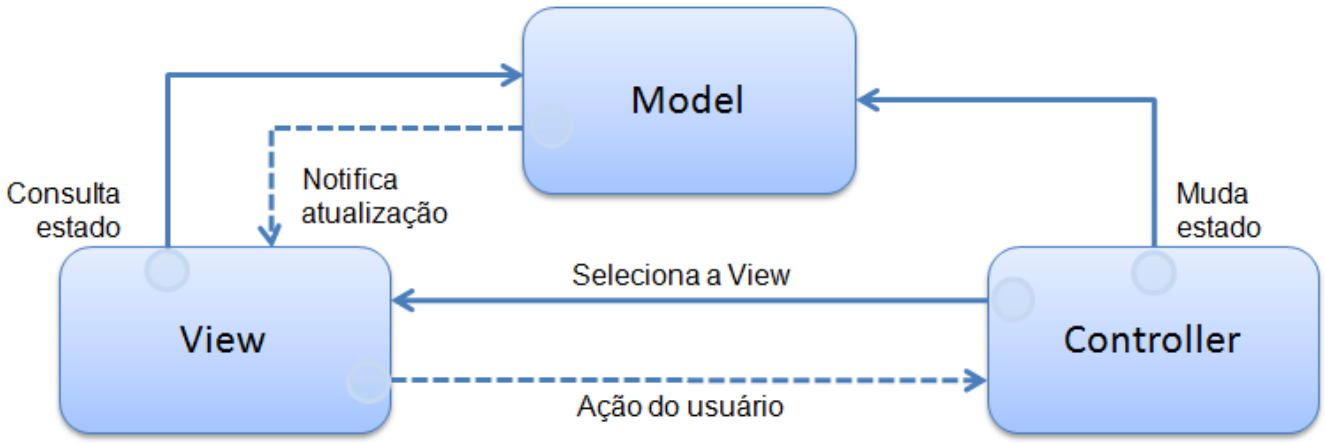
A arquitetura Model-View-Controller ou MVC, como é mais conhecida, é um padrão de arquitetura de software que visa separar a lógica de negócio da lógica de apresentação (a interface), permitindo o desenvolvimento, teste e manutenção isolada de ambos.
Aqueles que já desenvolveram uma aplicação em AdvPL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

2. Arquitetura MVC

A arquitetura **MVC**, 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.

A arquitetura MVC possui três componentes básicos:

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©



- **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 está 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.

2.1. 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**: Contém a construção e a definição do Model, lembrando que o Modelo de dados (Model) contém as regras de negócio;
- **ViewDef**: Contém a construção e definição da View, ou seja, será a construção da interface;
- **MenuDef**: Contém 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, está aplicação já terá disponível.

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

2.2. 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 possuir 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). ela se baseia em estruturas e essas por sua vez é que podem vir do metadados ou serem construídas manualmente. A ModelDef deve ser uma Static Function dentro da aplicação.

1.1. 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) recebendo 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 Business 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.

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

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:

- Título
- Nome da aplicação associada
- Reservado;
- 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;

3. 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.

3.1. 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.

- 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.

3.2. 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 obtida pelo metadados (dicionários).

```
oBrowse:SetAlias('SA1')
```

Definimos o título que será exibido como método SetDescription.

```
oBrowse:SetDescription('Cadastro de Cliente')
```

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.

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

3.2.1. 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 =='F'", "YELLOW", "Cons.Final"      )
oBrowse:AddLegend( "ZA0_TIPO=='L'", "BLUE"  , "Produtor Rural"  )
```

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
- **PINK:** Para a cor Rosa
- **WHITE:** Para a cor Branca

- **cDescrição:** a que será exibida para cada item da legenda

> - 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.

3.2.2. 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("A1_TIPO=='F'")
ou
oBrowse:SetFilterDefault( "Empty(A1_ULTCOM)" )
```

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).

O filtro da aplicação não poderá ser desabilitado pelo usuário

3.2.3. Desabilitar o 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()
```

3.3. 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('SA1')

// Definição da legenda
oBrowse:AddLegend("A1_TIPO='F'","YELLOW","Cons.Final")
oBrowse:AddLegend("A1_TIPO='L'","BLUE"  ,"Produtor Rural")
// Definição de filtro
oBrowse:SetFilterDefault( "A1_TIPO=='F'" )

// Titulo da Browse
oBrowse:SetDescription('Cadastro de Clientes')

// Opcionalmente pode ser desligado a exibição dos detalhes
```


Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

```
//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 contém as regras de negócio e a ViewDef que contém 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 uma 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.

4.1. Criando o MenuDef

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

```
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
```

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
```

- **TITLE:** nome do item no menu
- **ACTION:** 'VIEWDEF.nome_do_arquivo_fonte' PRW
- **OPERATION:** Valor que define a operação a ser executada(inclusão,alteração,etc)
- **ACCESS:** Valor que define o nível de acesso

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

```
Static Function MenuDef()
Return FWMVCMenu( 'COMP011_MVC' ))
```

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

4.2. Construção da função ModelDef

Nessa função são definidas as regras de negócio ou modelo de dados *(Model)*. Elas contêm as definições de:

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

Iniciamos a função **ModelDef**:

```
Static Function ModelDef()
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) não* pode ter o mesmo nome da função. Por exemplo, se estamos escrevendo a função XPTO, o identificador *(ID)* do modelo de dados *(Model)não* poderá ser XPTO.

4.2.1. 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.

- Estrutura dos Campos;
- Índices;
- Gatilhos;
- Regras de preenchimento;

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:

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

- **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 oStruSA1 := FWFormStruct( 1, 'SA1' )
```

No exemplo, o objeto oStruSA1 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 (SA1).

> 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 ambém os campos virtuais.
> Para a interface (**View**) a função **FWFormStruct**, traz para a estrutura os campos conforme o nível, uso ou módulo.

4.2.2. Criação de um componente de formulários (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( 'SA1MASTER', /*cOwner*/, oStruSA1 )
```

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

- **SA1MASTER:** é o identificador (ID) dado ao componente de formulário no modelo, **oStruSA1** 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*.

4.2.3. Criação de um componente de formulários na interface (AddField)

Adicionamos na 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_SA1', oStruSA1, 'SA1MASTER' )
```

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

VIEW_SA1 é o identificador (ID) dado ao componente da interface (View), **oStruSA1** é a estrutura que será usada e **SA1MASTER** 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 **SA1MASTER** serão exibidos na interface (View) no componente **VIEW_SA1**

4.2.4. 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 Cliente)
```

Adicionamos a descrição dos componentes do modelo de dados:

```
oModel:GetModel( 'SA1MASTER' ):SetDescription( 'Dados dos Clientes' )
```

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.

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

4.2.5. Indice do Model PrimaryKey

Atribui a PrimaryKey da entidade Modelo. A primarykey é imprescindivel para a correta operação das rotinas automáticas, web-services, relatórios, etc... Não é necessário informar uma para cada submodelo, a Primary Key é definida somente para o primeiro submodelo do tipo FormField.

```
oModel:SetPrimaryKey({ 'A1_COD', 'A1_LOJA', 'A1_FILIAL' })
```

4.2.6. Finalização de ModelDef

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

```
Return oModel
```

4.3. Exemplo completo da ModelDef

```
Static Function ModelDef()

Local oModel // Modelo de dados que será construído

// Cria o objeto do Modelo de Dados
oModel := MPFormModel():New('COMP011M' )

// Cria a estrutura a ser usada no Modelo de Dados
Local oStruSA1 := FWFormStruct( 1, 'SA1' )

// Adiciona ao modelo um componente de formulário
oModel:AddFields( 'SA1MASTER', /*cOwner*/, oStruSA1)

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

// Adiciona a descrição do Componente do Modelo de Dados
oModel:GetModel( 'SA1MASTER' ):SetDescription( 'Dados do Cliente' )

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

5. 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** contém 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 o acesso do modelo de qualquer outro fonte em MVC, com isso podemos reaproveitar o modelo de dados (Model) em mais de uma interface (View).

```
Local oModel := FWLoadModel('COMP011M')
```

- **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).

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

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

5.1. 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( '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

5.2. 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_SA1', 'TELA' )
```

Desta forma o componente *VIEW_SA1* será exibido na tela utilizando o box *TELA*.

5.3. Removendo campos da tela

Remoção de campos, são tratados na visão ViewDef com o metodo RemoveField No exemplo, coloco condição para não mostrar determinados campos

```
oStruct:= FWFormStruct(2, "SA1")

If __cUserID <> "000000"
    oStruct:RemoveField( "A1_DESC" )
Endif
```

5.4. Finalização da ViewDef

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

```
Return oView
```

5.5. 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 oStruSA1 := FWFormStruct( 2, 'SA1' )

// 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_SA1', oStruSA1, 'SA1MASTER' )

// 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_SA1', 'TELA' )

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

6. 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.**
FWLoadModel(<nome do fonte>)

Exemplo:

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

7. 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.**
FWLoadView (<nome do fonte>)

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

Exemplo:

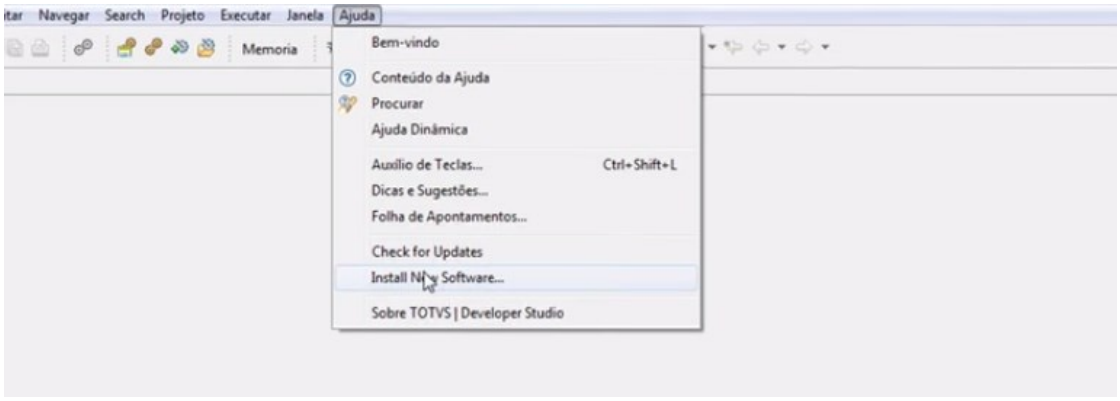
```
Static Function ViewDef()
// Utilizando uma view que ja existe em outra aplicacao
Return FWLoadView( 'COMP011_MVC' )
```

8. Intalação do Desenhador MVC

O **Desenhador MVC** é um conjunto de *plug-ins* que funcionam dentro do **TDS (Totvs Developer Studio)** sendo uma ferramenta de auxilio na criação e manutenção de códigos fontes escritos em AdvPL em conjunto com o framework de desenvolvimento.

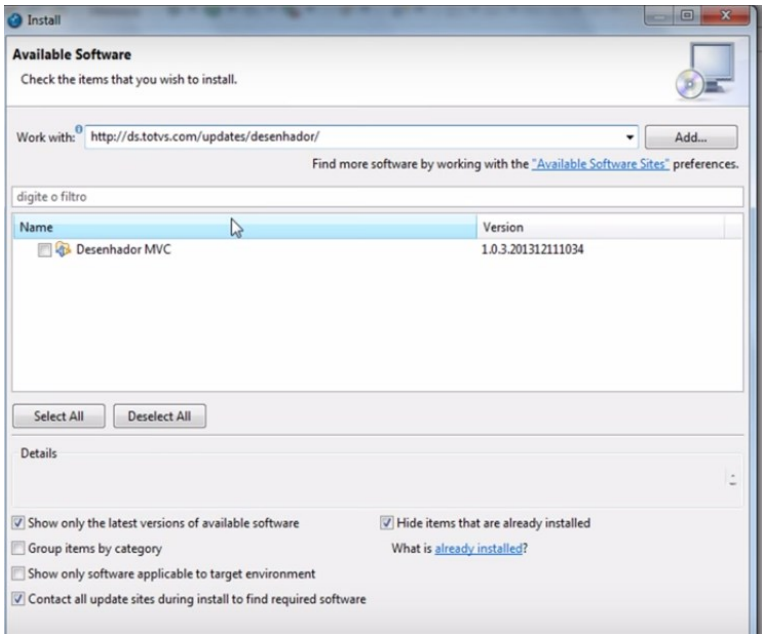
Para fazer a instalação do Desenhador MVC é necessário ter instalado o TDS.

Iremos fazer a instação do Plugin MVC. Ao acessar o TDS ir na opção Ajuda->Install New Software



Após acessar preencher Work With com endereço do instador do desenhador MVC:

http://ds.totvs.com/updates/desenhador/



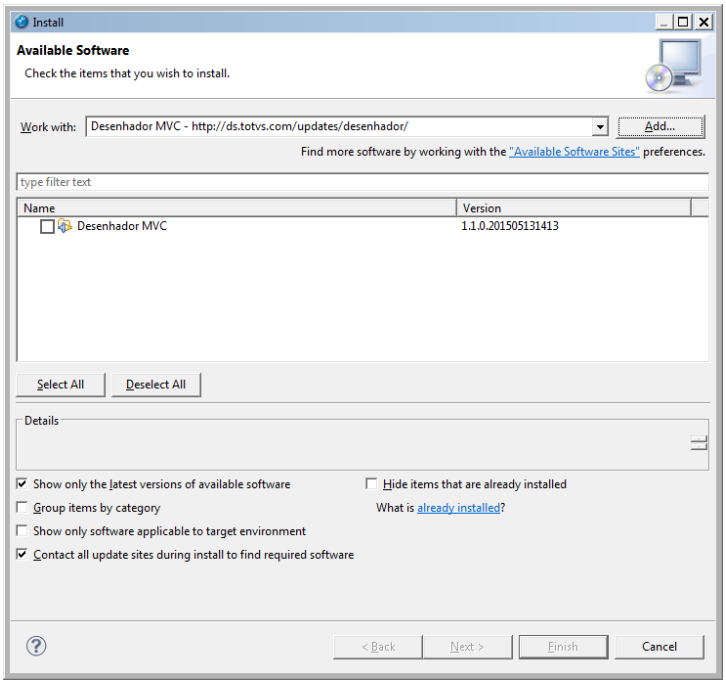
As opções tem que estar habilitadas:

- **Show Only the latest version of available software**
- **Contact all update sites during install ti find required software**

As demais opções desabilitada
Selecionar a opção:

- **Desenhador MVC**

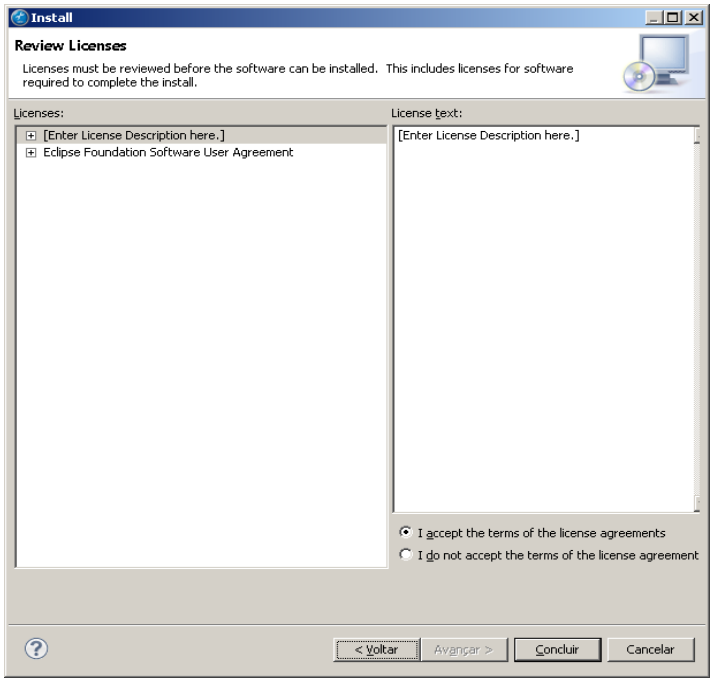
Após marcar a opção selecionar o botão Next.



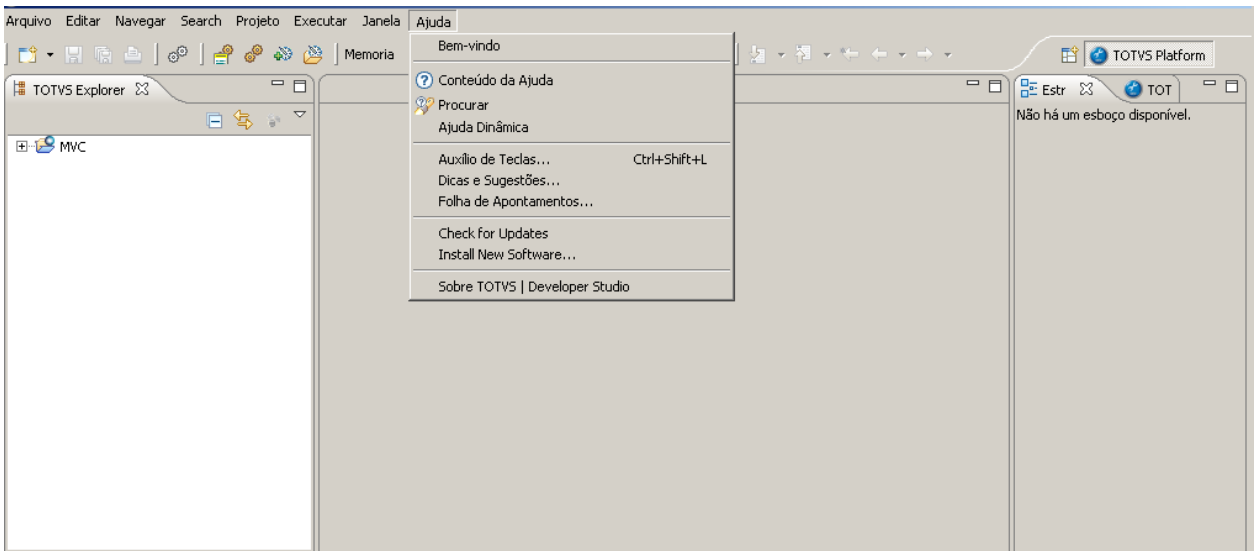
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

Seleicona a opção que aceita o termos de licença de uso.

- **I accept the berms of license agreements**

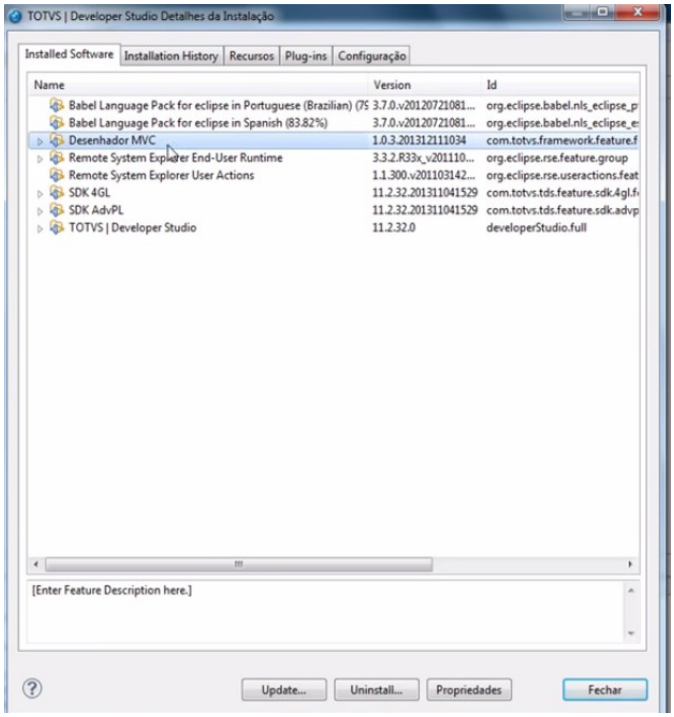


Selecionar o botão concluir, após esses processo a desenhador sera instalado,para confirmar a instalação do desenhador acessar Ajuda selecionar a opção Sobre Totvs | Developer Studio.



Selecionar o botão “Detalhes da Instação”.

Ira listar todos os plug-ins indatado no TDS, localizar o “Desenhador MVC”, caso não esteja na lista refazer o processo novamente caso ocorer algum erro na instalação contate o suporte da Totvs.



Pré-Requisitos instalação do desenhador:

- TDS atualizado
- Ambiente com LIB atualizada (Portal do Cliente)

Pré-Requisitos instalação do desenhador:

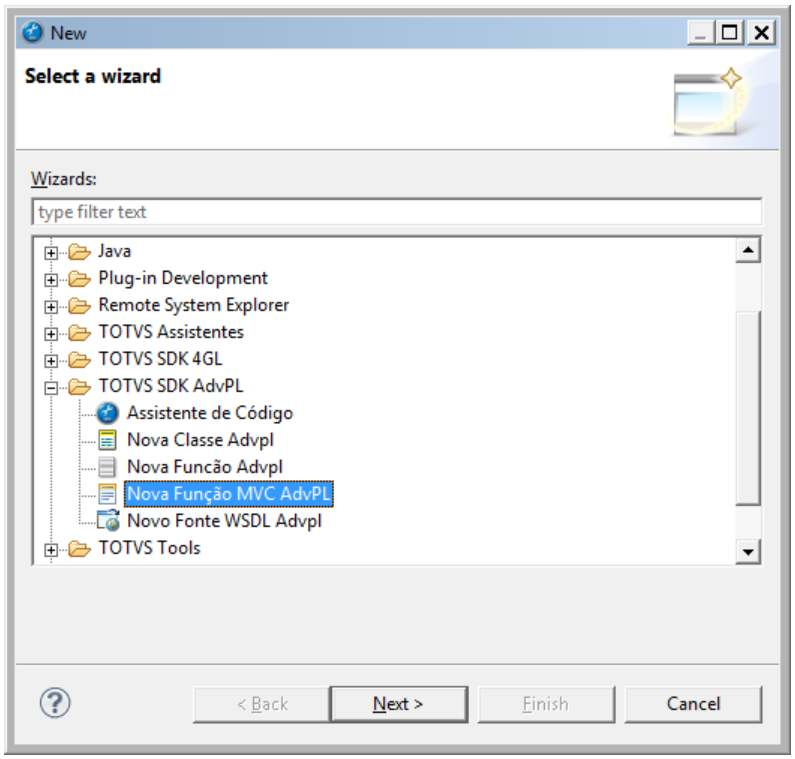
- TDS atualizado
- Ambiente com LIB atualizada (Portal do Cliente)

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

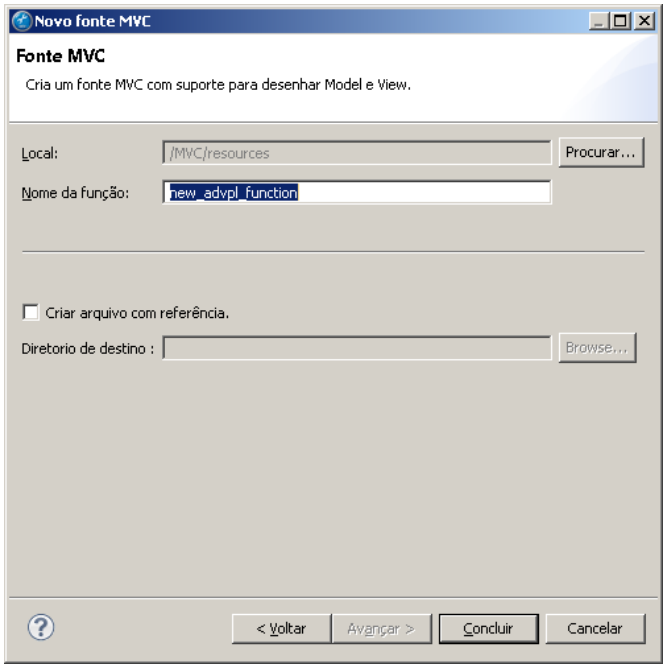
8.1. Criação de um novo fonte Desenhador MVC

Os fontes MVC AdvPL precisam necessariamente estar dentro de um projeto TOTVS.

Para criar um fonte novo, clique em Arquivos->Novo->Outras, escolha o item Nova Função MVC ADVPL do Totvs SDK AdvPL.

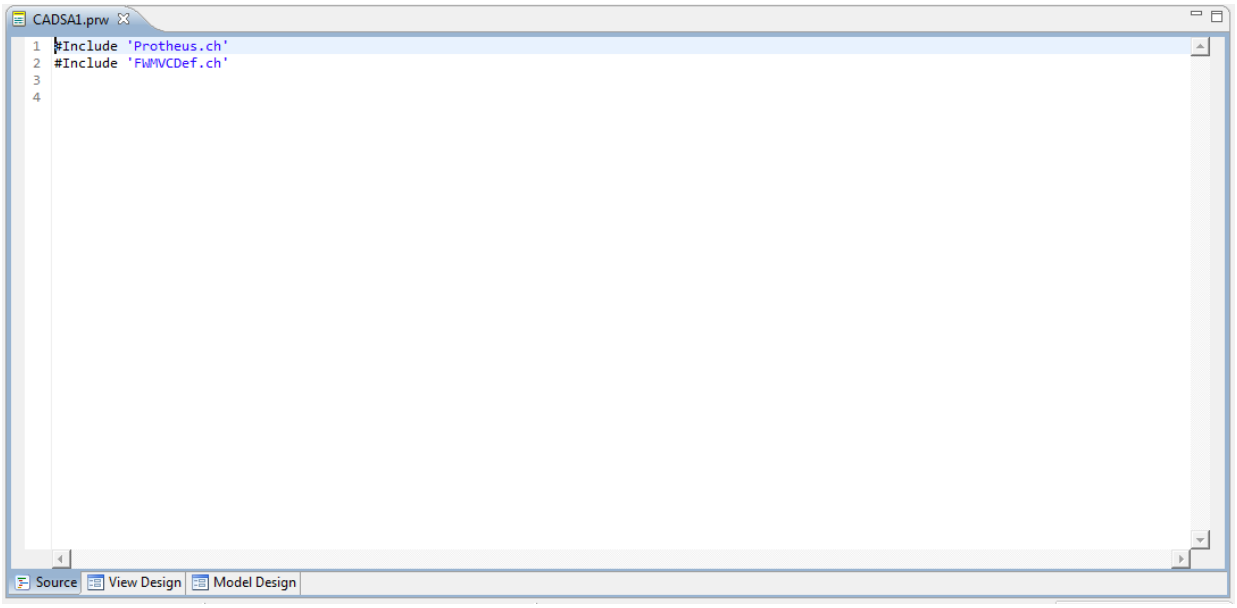


Na tela do novo fonte MVC, preencher os campos:



- **Local:** Informe a pasta do projeto que deseja salvar o fonte.
- **Nome da função:** Informe o nome do PRW**.**
- **Criar Arquivo com referência:** Para salvar o fonte fora do WorkSpace.

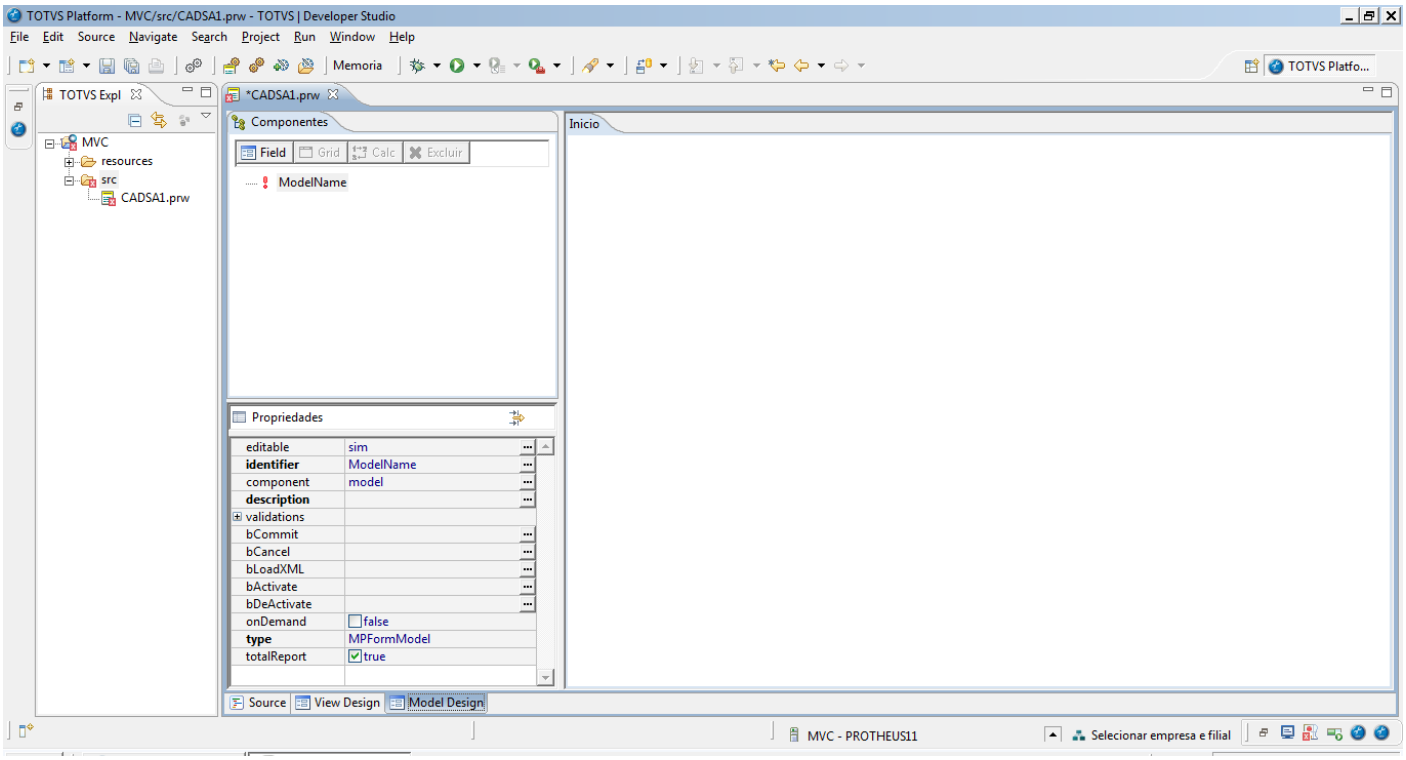
O Fonte é aberto com o Desenhador MVC possui três abas Source, View e Model.



Com novo plugin MVC possui novas validações do compilador facilitando o desenvolvimento.

Para criar uma estrutura MVC necessário criar Model onde iremos defir qual estrutura iremos trabalhar, selecionar a aba Model Design

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©



Na guia Model possui todos os componentes para criação do ModelDef:

Componentes:

Field – Adiciona no Model um sub-Modelo de campos “Entoiche”

- **Grid** - Adiciona no Model um sub-Modelo de grid “GetDados”
- **Calc** – Adiciona no grid campos totalizadores

Propriedades:

- **Editable:** Se o modelo for carregado usando a função FWloadModel não é possível alterar as propriedades
- **Identifier:** Identificador do Modelo
- **Component:** Nome do componente
- **Description:** Atribui ao modelo um texto explicativo sobre o objetivo do Modelo. O objetivo é mostrado em diversos operações, tais como web services, relatórios e schemas (xsd).
- **Validations:**

  bPre: Bloco de código de pré-validação do modelo. O bloco recebe como parametro o objeto de Model e deve retornar um valor lógico. Quando houver uma tentativa de atualização de valor de qualquer Submodelo o bloco de código será invocado. Caso o retorno seja verdadeiro, a alteração será permitida, se retornar falso não será possível concluir a alteração e um erro será atribuido ao model.

  bPos: Bloco de código de pós-validação do modelo, equilave ao "TUDOOK". O bloco recebe como parametro o objeto de Model e deve retornar um valor lógico. O bloco será invocado antes da persistência dos dados para validar o model. Caso o retorno seja verdadeiro e não haja nenhum submodelo invalido, será feita a gravação dos dados. Se retornar falso não será possível realizar a gravação e um erro será atribuido ao model.

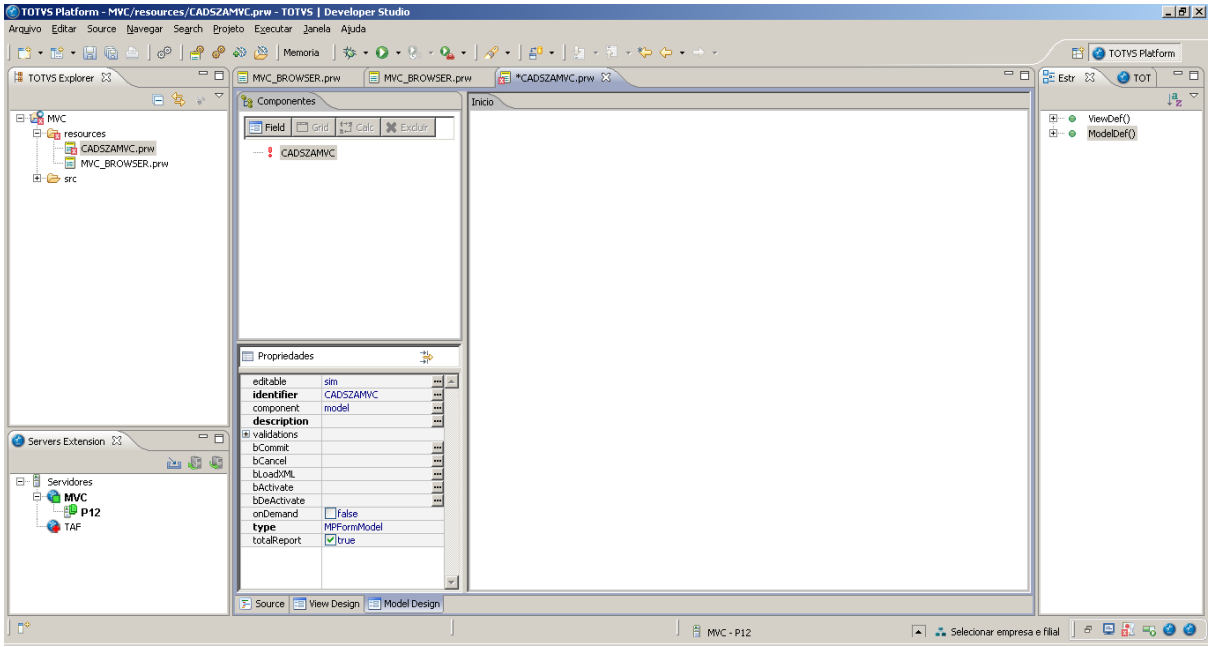
  bVldActivate: Bloco de código que será chamado antes do Activate do model. Ele pode ser utilizado para inibir a inicialização do model. Se o retorno for negativo uma exceção de usuário será gerada. O code-block recebe como parametro o objeto model.
- **bCommit::**Bloco de código de persistência dos dados, ele é invocado pelo método CommitData. O bloco recebe como parametro o objeto do Model e deve realizar a gravação dos dados.

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

- **bCancel:** Bloco de código de cancelamento da edição, ele é invocado pelo método CancelData. O bloco recebe como parametro o objeto do Model.
- **bLoadXML:** Configura o modelo para ser ativado com uma folha de dados em XML. Bloco de código recebe o modelo por parametro e deve retornar um XML com os dados baseado no modelo.
- **bActivate**: Bloco de código que será chamado logo após o Activate do model. Esse bloco recebe como parametro o proprio model.
- **bDeActivate:** Bloco de código que será chamado logo após o DeActivate do model. Esse bloco recebe como parametro o proprio model.
- **onDemand:** Define se a carga dos dados será por demanda. Quando o model é por demanda, os dados do submodelo não serão carregados de uma vez no activate. Conforme eles forem manipulados eles serão carregados. O método é útil em modelos que contem muitos submodelos e que podem ficar lentos devido a grande quantidade de dados.
- **Type:** A opção MPFormModel realiza tratamento nas variaveis de memória e possibilita o uso do Help para as mensagens. A opção FWFormModel não realiza esses tratamentos, por isso é necessário utilizar o SetErrorMessage em todas as mensagens de erro. A opção mais utilizada é a MPFormModel.
- **TotalReport:** Configura se o relatório gerado pelo metodo ReportDef deverá totalizar todos os campos numéricos.

Para criar a estrutura básica do Model necessário informar os campos:

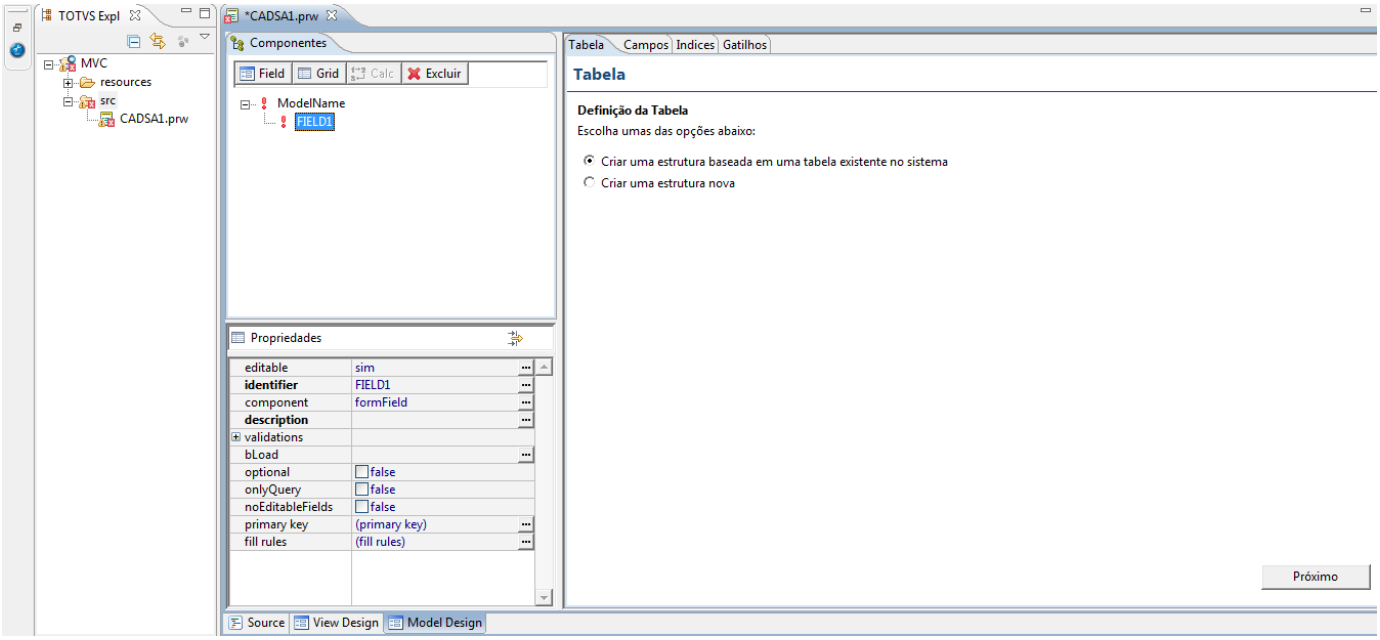
- **Identifier:** Nome do Model, o identificador (ID) do modelo de dados (Model) pode ter o mesmo nome da função principal e esta prática é recomendada para facilitar a codificação.
- **Component:** Nome do componente
- **Description:** Adicionamos a descrição ao modelo de dados (Model). O objetivo é mostrado em diversos operações, tais como web services, relatórios e schemas (xsd).



Após criar estrutura do modelo de dados (Model), necessario adicionar um componente de formulário, iremos criar no exemplo o componente “Field.

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

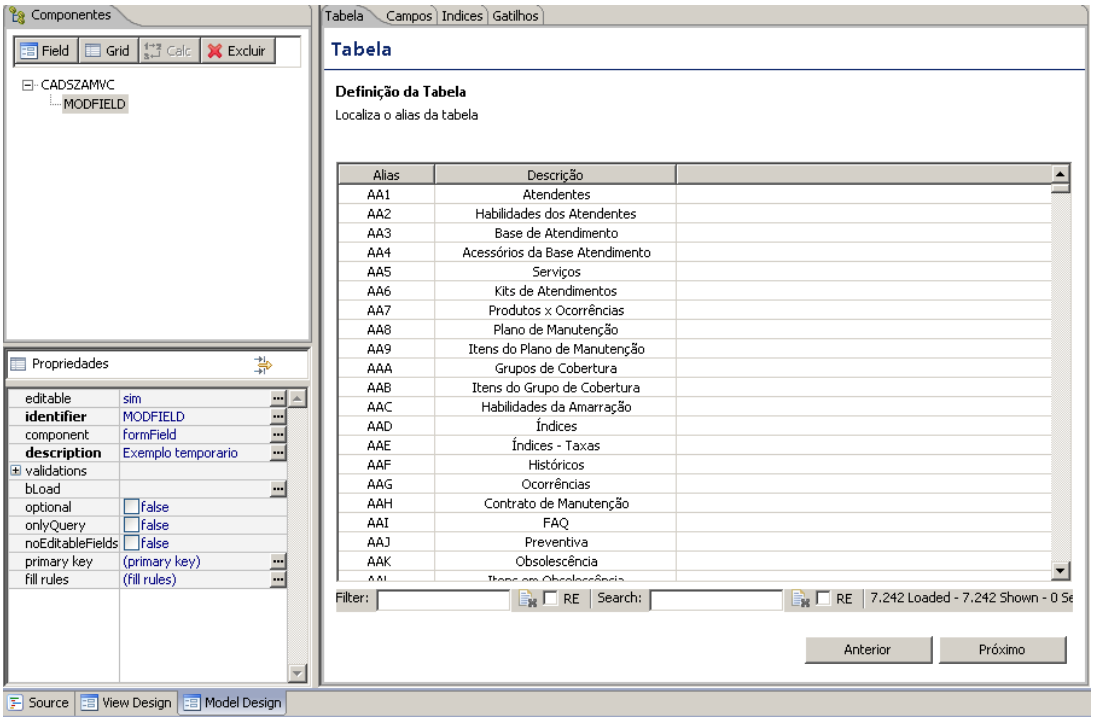
Para criar o componente selcionar na guia de componentes a opção Field, após selecionar o componete, ira aparecer dentro da estrutura do Model.



Nas propriedade dos componente possui a estrutura:

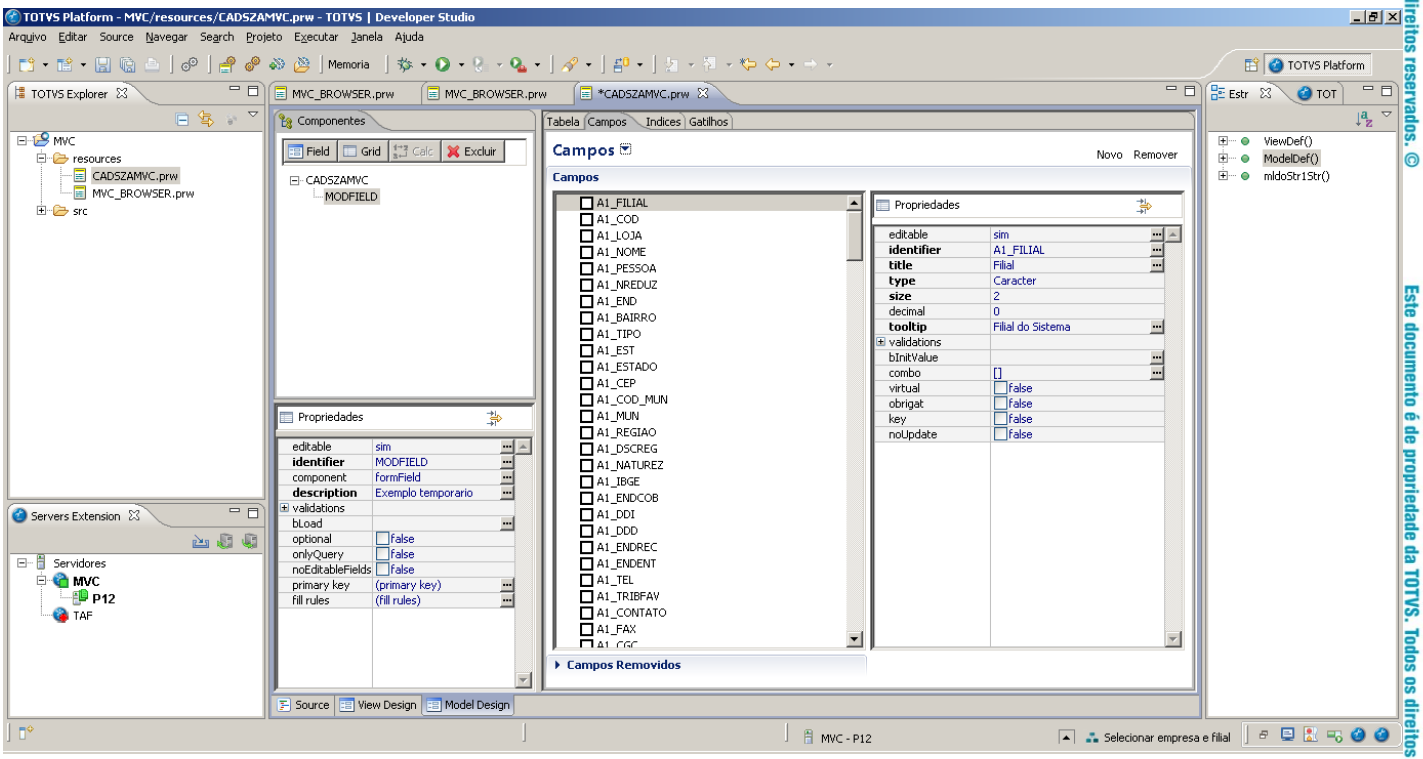
- **Definições da Tabela:**
  - **Cria a estrutura a ser usada no Modelo de Dados:** Lista as tabelas existentes no SX2
  - **Cria uma uma estrutura nova:** Criar estrutura de tabela temporária

Na guia tabela ira lista todos os dados do arquivo SX2.



Para relacionar a tabela selecionar na lista a tabela desejada.

Na pasta "**Campos**" ira lista os campos com as propriedades dele, podemos interagir nos dados, mas não altera a estruro no SX3.

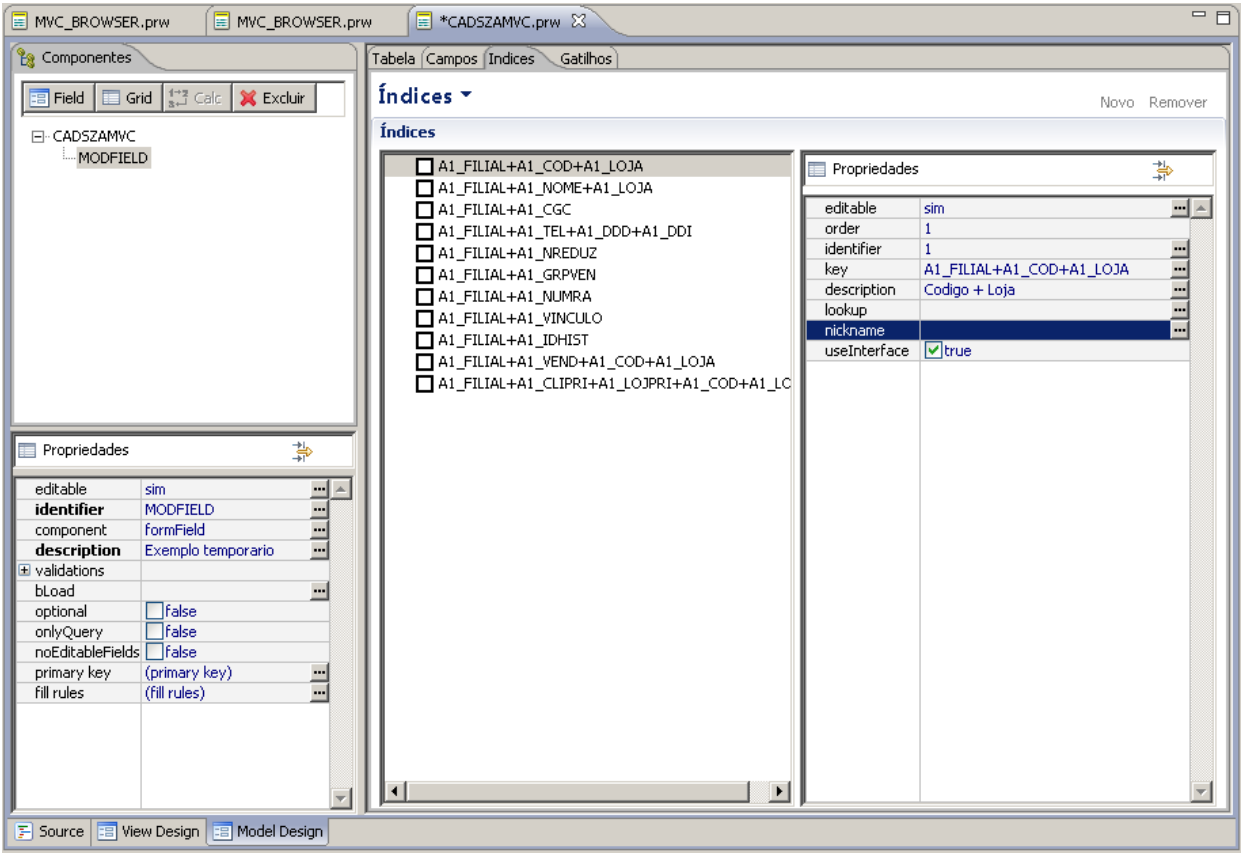


Todos os campos na lista ira aparecer na estrutura

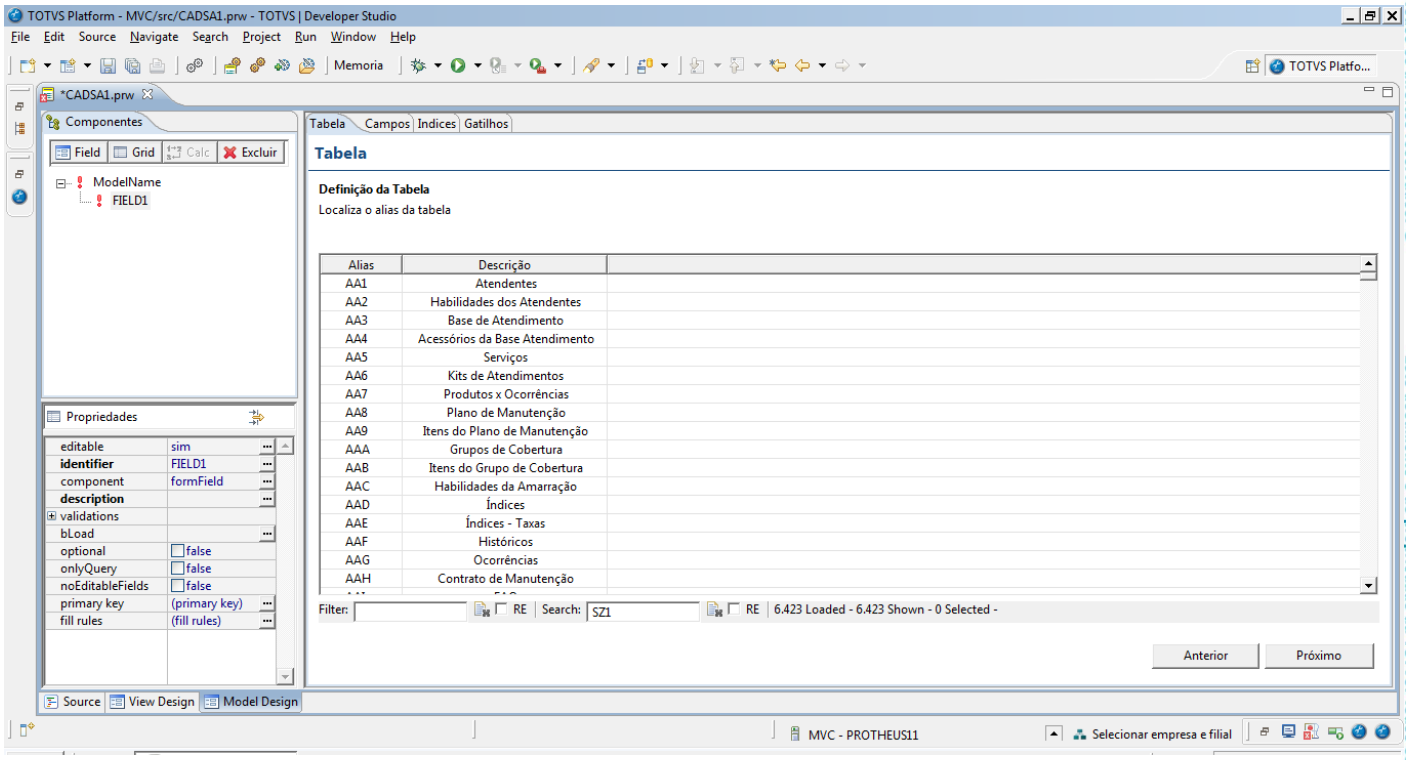
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

È possível criar novos campos temporário selecionando o botão novo necessário estar no MODEL e na VIEW.

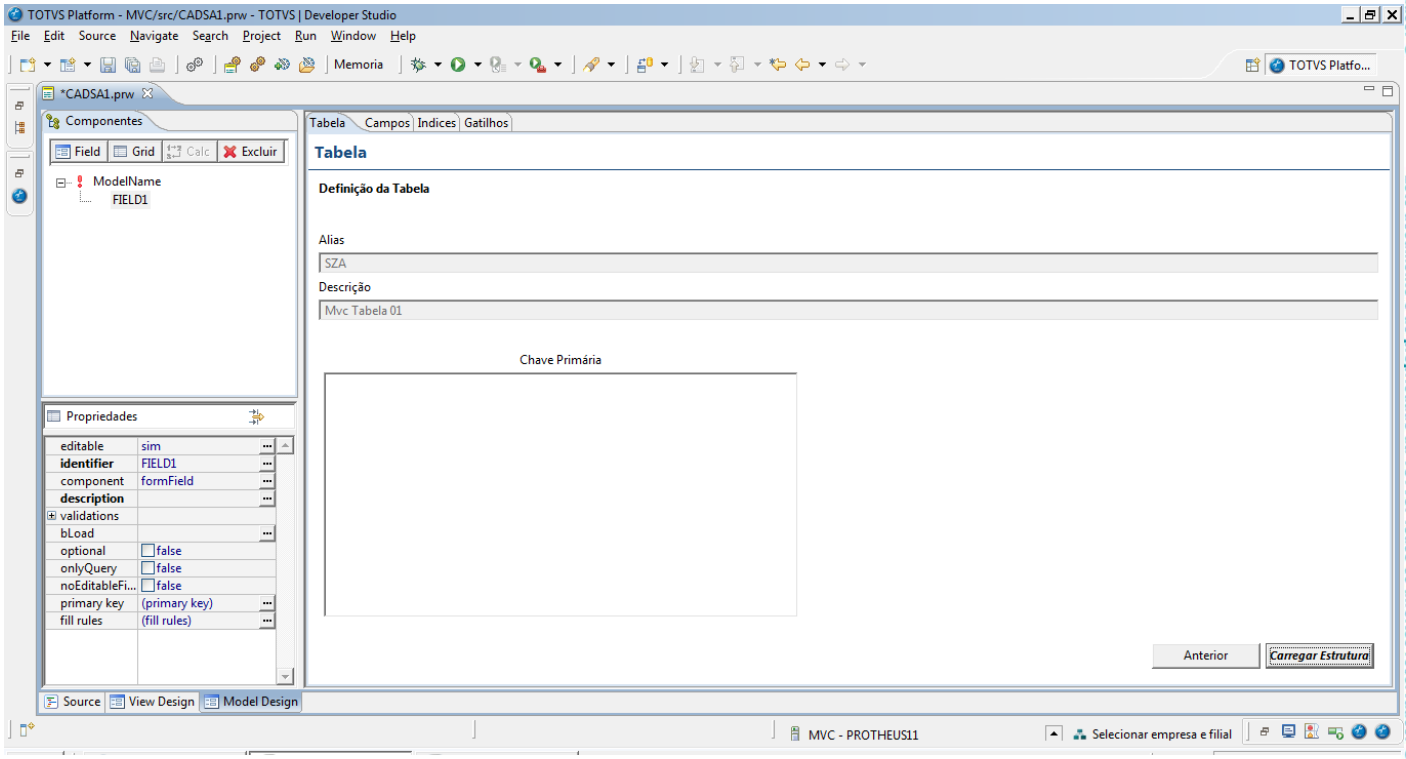
Na pasta Indices, lista os dados da SIX com as suas propriedades



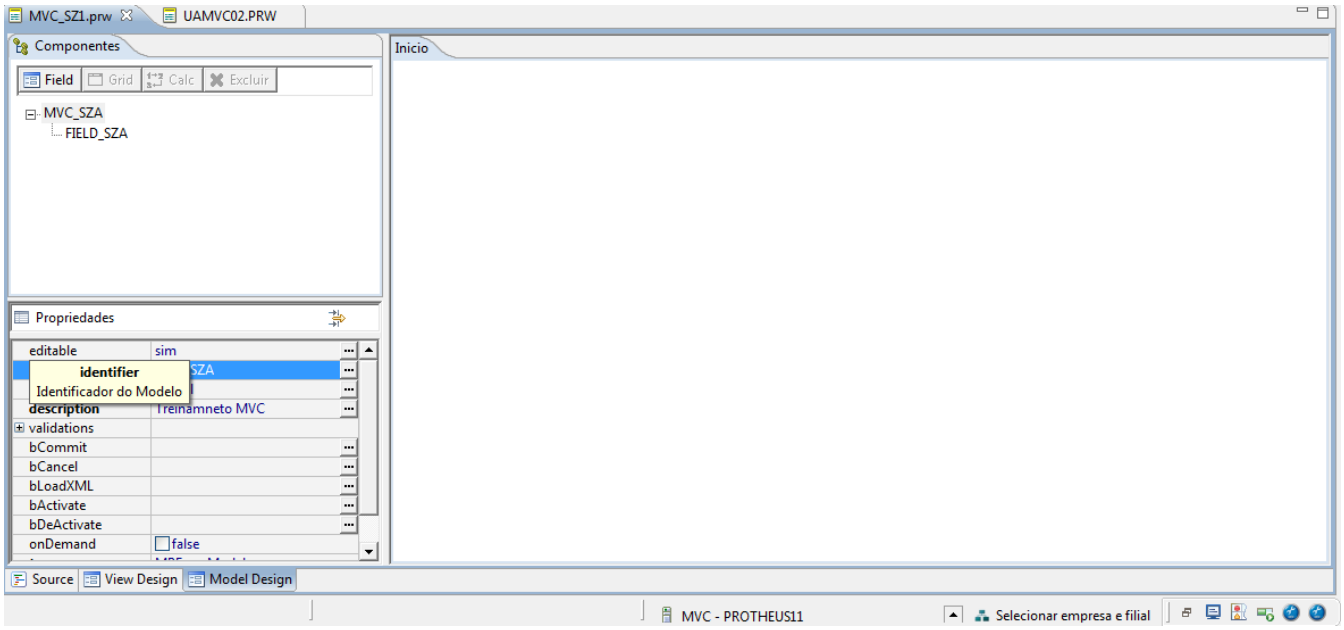
Ira listar todas as tabelas do SX2 no grid, selecionar a tabela que irá trabalhar, após informar a tabela selecionar o botão **Proximo**



Ao informar a tabela, selecionar o botão "**Carregar Estrutura"**, todos os campos, índices e gatilhos da tabela foram carregados para a estrutura de dados.

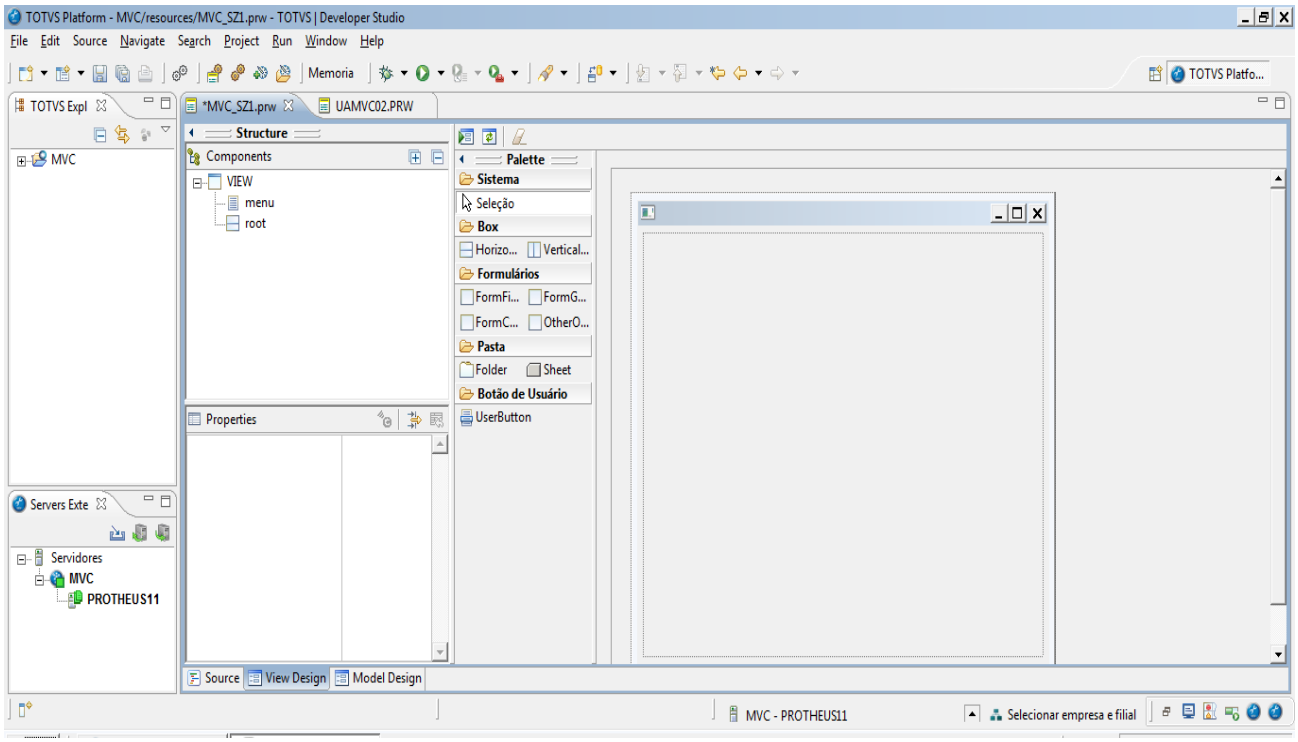


Na Guia de propriedades você tem todas as opções **Model**, ao passar o mause sobre as propridedades exemplo: "**Identifier**", sera aberto uma ajuda com as explicações do bloco de persistência dos dados.



Após criar Model iremos defir as propriedades do View, selecionar no fonte MVC a aba View Design

Ira listas todas as probriedades para definir a função ViewDef.



Na árvore de componentes:

- **View:** Estrutura do View
- **Menu:** Botões no View ira listar todo os botões criado pelo “UserButton”
- **Root:** Lista todos os componentes visuis
- **Menu:** É um componente onde serão colocados os botões de usuário.
- **Root:** É o componente base onde os componentes de interface serão alocados.
- **Palette:** Possui todos os componente de visualização do Model

Palette

- **Box:** Caixa de visualização do Model

  HorizontaBox:O Box dever ser utilizado para dividir a tela. O Box Horizontal divide a tela na posição horizontal e todos os componentes que forem criados dentro dele tambem assumirão essa orientação. Atenção: Esse box somente pode ser colocado dentro de um box vertical.

  VerticalBox: Box é um componente utilizado para dividir a tela. O Box Vertical divide a tela na posição vertical e todos os componentes que forem criados dentro dele também assumirão essa orientação. Atenção: Esse box somente pode ser colocado dentro de um box horizontal.
- **Formulários:**

  FormField : Formulário que exibe campos no mesmo modelo da antiga enchoice. Para usar esse componente se faz necessário criar um submodelo do tipo field no Model e associar o mesmo ao formulário.

  FormGrid: Formulário que exibe um grid, ele mostra diversos registros na tela. Para usar esse componente se faz necessário criar um submodelo do tipo grid no Model e associar o mesmo ao formulário.

  FormCalc: Formulário que exibe campos calculados criados no Model. Para usar esse componente se faz necessário criar um submodelo do tipo calc no Model e associar o mesmo ao formulário.

  OtherObject:: Cria um painel onde pode ser colocado qualquer coisa que o desenvolvedor desejar. Para criar as coisas use o bloco de código de activate do componente. Atenção: Tudo que for criado dentro desse componente NÃO será renderizado nesse plugin.
- **Pasta:**

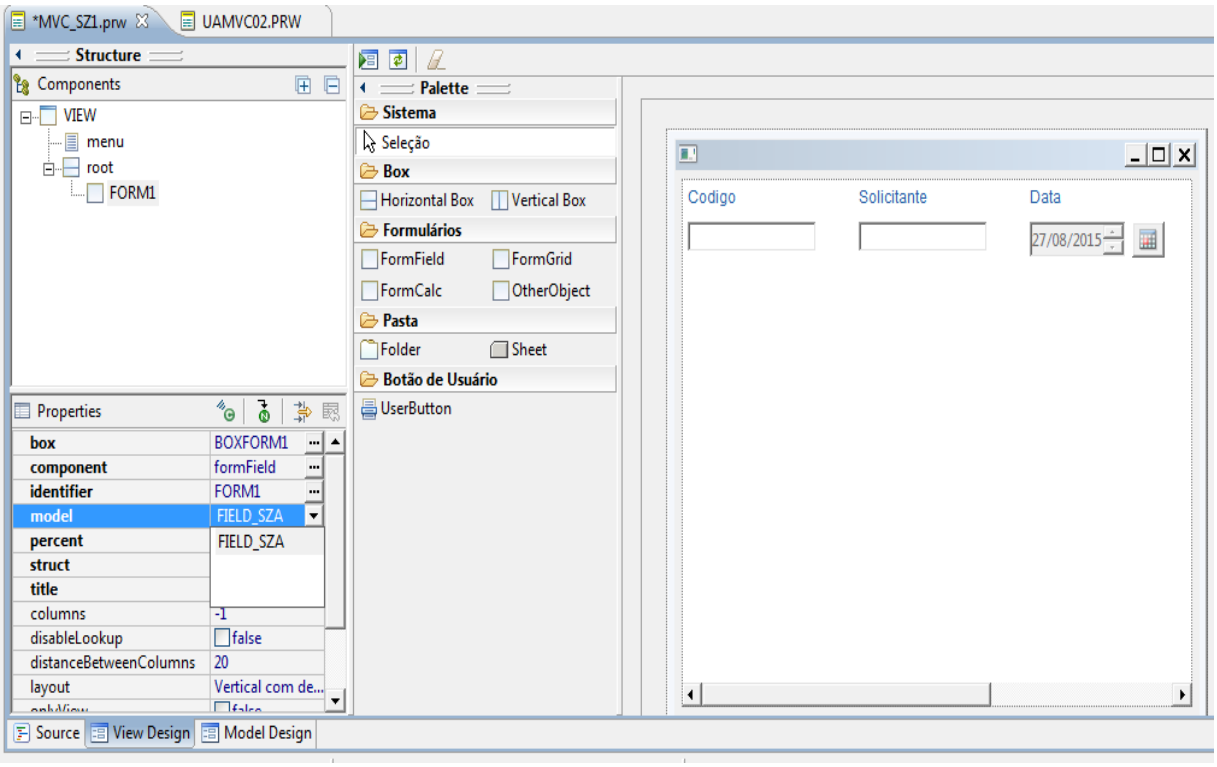
  Folder: Cria um componente de pasta que é capaz de armazenar dentro dele diversas abas

  Sheet: Cria uma aba dentro de uma Folder. Esse componente permite criar novas divisões dentro da aba, criando diversas opções na tela.
- **Botão de Usuário:**

  UserButton: Adiciona botões do desenvolvedor na barra de ferramentas do formulário.

Para criar a ViewDef necessário criar um componente de visualização, selecionar o componente “FormField” e adicionar no componte “root”, em propriedades “Model” informar a identificador do Model criado.

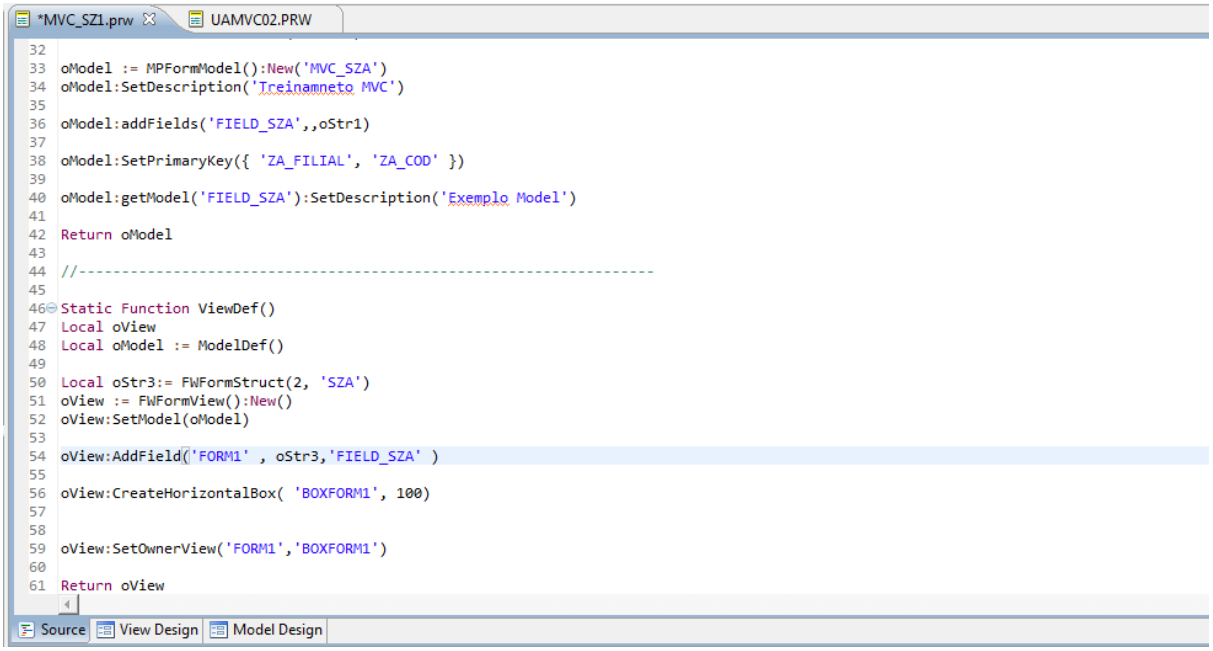
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©



Acabamos de criar um fonte em MVC com informações básicas

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

Tudo que foi desenvolvido nas guias View e Model criou o fonte na guia Source



9. 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.

9.1. Mensagens exibidas na interface

As mensagens são usadas principalmente durante as validações feitas no modelo de dados. 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 camada modelo de dados.
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 irá 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( ,, ' Título do campo','', ' Mensagem a ser exibida no help.', 1, 0 )
EndIf
```


Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

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.

9.2. 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(<cActionIID>, <bAction>)

Onde:

- **cActionIID**: 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 ) } )
```

Dica

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.

9.3. 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**.

```
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 ) } )
```

9.4. 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.**

Para acasser a operção necessario esta com o Model

Local nVar := oModel:GetOperation()

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

- **MODEL_OPERATION_INSERT**
- **MODEL_OPERATION_UPDATE**
- **MODEL_OPERATION_DELETE**

Exemplo:

```
 If oModel:GetOperation() == MODEL_OPERATION_INSERT
      Help( ,, 'Help',, 'Não permitido incluir.', 1, 0 )
EndIf
```


Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

9.5. Obtenção de componente do modelo de dados (GetModel)

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

```
Local oModel := oModel:GetModel( 'NOME DO ID DO MODELO DE DADOS' )
```

Podemos capiturar o modelo completo.

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

9.6. 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) consegue acessar as estruturas dos campos podendo capturar ou manupular o dado no (Model).

```
cVar := oModel:GetValue( 'ID do componente', 'Nome do campo para obter o dado’)
```

Ou em casos que não tenha recuperado o modelo da tabela em uma variável:

```
cVar := oModel:GetModel('ID'):GetValue(‘Nome do campo do qual se deseja obter o dado’) )
```

- **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( ‘ID’, 'Nome do Campo', 'Valor para o campo' )
```

Ou em casos que não tenha recuperado o modelo da tabela em uma variável:

```
oModel:SetValue('Nome do Campo', 'Valor para o campo' )
```

9.7. Adicionando botão na tela

Para adicionar botões na barra de ferramentas do formulário necessário utilizar a função:

- FWFORMVIEW():addUserButton(<cTitle >, <cResource >, <bBloco >, [cToolTip], [nShortCut], [aOptions])

Exemplo:

```
oView:AddUserButton('NomeBotão','CLIPS',;
{|| fValSto()},'NomeBotao',/*nShortCut*/,;
{MODEL_OPERATION_INSERT, MODEL_OPERATION_UPDATE})
```

10. Manipulação da componente FormGrid

Alguns tratamentos que poderá ser feito nos componentes de *grid* de um modelo de dados *(Model)*

10.1. 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.

- `oModel:SetRelation( 'FMRDETAIL', { { 'C6_FILIAL', 'xFilial( "SC6" )' }, ;`
  `{ 'C6_NUM', 'C5_NUM' } }, SC6->( IndexKey( 1 ) ) )`

- **FMRDETAIL**: é o identificador (ID) da entidade Detail, o segundo parâmetro é um vetor bidimensional 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.

O relacionamento sempre é definido do Detail (Filho) para o Master (Pai), tanto no identificador (ID) quanto na ordem do vetor bidimensional.

10.2. 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.

```
oView:AddIncrementField( 'VIEW_SC6', 'C6_ITEM' )
```


Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

Onde **VIEW_SC6** é o identificador (ID) do componente da interface (View), onde se encontra o campo e **C6_ITEM** o nome do campo que será incrementado.

10.3. 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.

```
For nI := 1 To oModel: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 := oModel:Length( .T. ) // Quantidade linhas não apagadas
```

10.4. 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) está 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 no grid. Retornando, .T. (verdadeiro) a linha foi inserida.

```
Static Function COMP23ACAO()
Local oModel := FWModelActive()
Local oModelZA2 := oModel:GetModel( 'ZA2DETAIL' )
Local nI := 0
Local nCtInc := 0
Local nCtAlt := 0
Local nCtDel := nCtInc := nCtAlt := 0
Local aSaveLines := FWSaveRows()

For nI := 1 To oModel:Length()

      oModel:GoLine( nI )

      If oModel:IsDeleted()
       nCtDel++
```

```
      ElseIf oModel:IsInserted()
         nCtInc++
      ElseIf oModel:IsUpdated()
       nCtAlt++
      EndIf
Next
FwRestRows(aSaveLines)
```

10.5. 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 oModel: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.

10.6. 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 oModel := oModel:GetModel( 'ZA2DETAIL' )
Local nI := 0

For nI := 1 To oModel:Length()
  oModel:GoLine( nI )
  If !oModel:IsDeleted()
    oModel: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).

10.7. Recuperando uma linha da grid que está apagada

Utilizamos o método UnDeleteLine.

```
Local oModel := oModel:GetModel( 'SA1DETAIL' )
Local nI := 0
For nI := 1 To oModelSA1:Length()
```


Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

```
  If oModel:IsDeleted()
    oModel:GoLine( nI )

      oModel: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).

10.8. 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.

```
Local aSaveLines := FWSaveRows()
```

Antes do Return para restaurar o valor:

```
FwRestRows( aSaveLines )
```

10.9. Criação de pastas (CreateFolder)

Em MVC podemos criar pastas onde serão colocados os componentes da interface (View). Para isso utilizamos o método CreateFolder.

```
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.

```
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_SB1' , 'SUPERIOR' )
oView:SetOwnerView( 'VIEW_SB5' , 'INFERIOR' )
```

10.10. 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)

- **cId:** Identificador do componente de cálculos;
- **cOwner:** identificador do componente superior (owner). Não necessariamente é o componente de grid de onde virão os dados.
- **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) 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.

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

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

- **nTamanho:** tamanho do campo calculado (Se não for informado usa o tamanho padrão).
- **SUM:** será o tamanho do campo do componente de grid + 3;
- **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;

Atenção:

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

Para adicionar contador de registro, adicione a seguinte linha no ModelDef:

```
oModel:AddCalc(  'COUNTTOTAL',  'MODEL_AAA',  'MODEL_BBB',  'AAA_ID',  'Total',
'COUNT' )
```

- **COUNTTOTAL:** é o identificador do componente de cálculos
- **MODEL_AAA:** é o identificador do componente superior (*owner*)
- **MODEL_BBB:** é o código do componente de grid de onde virão os dados
- **AAA_ID:** é o nome do campo do componente de grid a que se refere o campo calculado
- **Total:** é o Identificador (nome) para o campo calculado
- **COUNT:** é o Identificador da operação a ser realizada

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**.

```
Local oCount := FWCalcStruct( oModel:GetModel('COUNTTOTAL') )
oView:AddField( 'VCALC_ID', oCount, 'INFERIOR' )
oView:CreateHorizontalBox( 'INFERIOR', 24)
oView:SetOwnerView('VCALC_ID','INFERIOR')
```

10.11. Outros objetos (AddOtherObjects)

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**

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

- **Id:** parâmetro é o identificador (ID)
- **Código de bloco:** para a criação dos outros objetos. 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 ) } )
```

Para o botão parecer na camada viewr necessário associa ao box que ira exibir os outros objetos

```
oView:SetOwnerView("OTHER_PANEL",'EMBAIXODIR')
```

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

```
Static Function COMP23BUT ( oPanel )
        TButton():New( 020, 020, "TESTE", oPanel, {|| MsgInfo('TESTE') }, 030, 010, , , .F., .T., .F., , .F., , , .F. )
Return( Nil )
```

10.12. 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.

- **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

- **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.*

- **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.*

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

10.12.1. 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.

No exemplo o campo **B1_COD** não poderá ter seu conteúdo repetido no *grid*. Também pode ser informado mais de um campo, criando assim um controle com chave composta.

```
oModel:GetModel( 'SB1DETAIL' ):SetUniqueLine( { 'B1_COD' } )
```

10.12.2. SetDeActivate

Antes da ativação do Model é permite executar função para validar o modo de operação, será chamado logo após o DeActivate do model. Esse bloco recebe como parametro o proprio model.

- oModel:SetDeActivate (bBloco())

Exemplo:

```
oModel:SetDeActivate   ({|oModel   |   MsgInfo("SetDeActivate","Model")   })
```

10.12.3. Pré-validação do Modelo

O bloco recebe como parametro o objeto de Model e deve retornar um valor lógico.
Quando houver uma tentativa de atualização de valor de qualquer Submodelo o bloco de código será invocado. Caso o retorno seja verdadeiro, a alteração será permitida, se retornar falso não será possível concluir a alteração e um erro será atribuido ao model, sendo necessário indicar a natureza do erro através da função Help.

- oModel:New(<cID >, <**bPre** >, <bPost >, <bCommit >, <bCancel >)

Exemplo:

```
Static Function ModelDef()

Local bPre := {|   | ValidPre(oModel)}
Local oModel := MPFormModel():New('SA1MODEL', , bPre).

Static Function ValidPre(oModel)

Local nOperation := oModel:GetOperation()
Local lRet := .T.

If nOperation == MODEL_OPERATION_UPDATE
   If Empty( oModel:GetValue( 'SA1MASTER', 'A1_CGC' ) )
      Help( ,, 'HELP',, 'Informe o CNPJ', 1, 0)
```

```
      lRet := .F.
   EndIf
EndIf

Return lRet
```

10.12.4. Pós-validação do Modelo

A validação pós-validação do modelo, equilave ao "TUDOOK". O bloco recebe como parametro o objeto de Model e deve retornar um valor lógico. O bloco será invocado antes da persistência dos dados para validar o model.
Caso o retorno seja verdadeiro e não haja nenhum submodelo invalido, será feita a gravação dos dados.
Se retornar falso não será possível realizar a gravação e um erro será atribuido ao model, sendo necessário indicar a natureza do erro através da função Help.

- MPFORMMODEL():New(<cID >, <bPre >, <**bPost** >, <bCommit >, <bCancel >)

Exemplo:

```
Static Function ModelDef()
Local oModel := MPFormModel():New('SA1MODEL', , { |oMdl| COMP011POS( oMdl ) 
})
Return

Static Function COMP011POS( oModel )
Local nOperation := oModel:GetOperation()
Local lRet := .T.

If nOperation == MODEL_OPERATION_UPDATE
   If Empty( oModel:GetValue( 'SA1MASTER', 'A1_COD' ) )
      Help( ,, 'HELP',, 'Informe o codigo do cliente', 1, 0)
      lRet := .F.
   EndIf
EndIf

Return lRet
```

10.12.5. 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.

- MPFORMMODEL():New(<cID >, <bPre >, <bPost >, <**bCommit** >, <bCancel >)

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 )
Local lGravo

lGravo := FWFormCommit( oModel )

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

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.

10.12.6. Cancelamento da gravação de dados

Essa validação realiza os tratamentos necessários ao cancelamento dos formulários de edição.O bloco recebe como parametro o objeto do Model.
Quando esse bloco é passado o tratamento de numeração automatica não é mais realizado, a menos que o bloco chame a função FWFormCancel.

- MPFORMMODEL():New(<cID >, <bPre >, <bPost >, <bCommit >, <**bCancel** >)

Exemplo:

```
oModel := MPFormModel():New("MODELO",,, ,{ | oModel | FWFormCancel(oModel). })
```

10.12.7. SetVldActivate

Será chamado antes do Activate do model. Ele pode Ser utilizado para inibir a inicialização do model. Se o retorno for negativo uma exceção de usuário será gerada. O code-block recebe como parâmetro o objeto model.

- MPFORMMODEL():SetVldActivate(<bBloco >)

10.13. Validações AddFields

Um submodelo do tipo Field permite manipular somente um registro por vez. Ele tem um relacionamento do tipo 1xN ou 1x1 com outros SubModelos ou então não tem nenhum relacionamento.

- FWFORMMODEL():AddFields([cId], [cOwner], [oModelStruct], [bPre], [bPost], [bLoad])-> NIL

10.13.1. Estruta Pré Validação do AddFilds

É invocado quando há uma tentativa de atribuição de valores. O bloco recebe por parametro o objeto do FormField(FWFormFieldsModel), a identificação da ação e a identificação do campo que está sofrendo a atribuição. As identificações que podem ser passadas são as seguintes:

- "CANSETVALUE" : valida se o submodelo pode ou não receber atribuição de valor.
- "SETVALUE" : valida se o campo do submodelo pode receber aquele valor.

Nesse caso o bloco recebe um quarto parametro que contem o valor que está sendo atribuido ao campo. Para todos os casos o bloco deve retornar um valor lógico, indicando se a ação pode ou não ser executada. Se o retorno for falso um erro será atribuido no Model, sendo necessário indicar a natureza do erro através do método SetErrorMessage.

- FWFORMMODEL():AddFields([cId], [cOwner], [oModelStruct], [**bPre**], [bPost], [bLoad])-> NIL

Exemplo:

```
Static Function ModelDef()
Local oModel
Local oStruZA1:= FWFormStruct(1,'SA1')
Local bPre := {|oFieldModel, cAction, cIDField, xValue|
validPre(oFieldModel,cAction,;
cIDField, xValue)}

oModel := FWFormModel():New('COMP021')
oModel:addFields('ZA1MASTER',,oStruZA1,bPre,bPos,bLoad)

Static Function validPre(oFieldModel, cAction, cIDField, xValue)
Local lRet := .T.

If cAction == "SETVALUE" .And. cIDField == "A1_CGC"
   Help( ,, 'HELP',, ('Não possível atribuir valor ao campo ' + cIDField) , 1,
0)
   lRet := .F.
EndIf

Return lRet
```

10.13.2. Estruta Pós Validação do AddFilds

É equivalente ao "TUDOOK". O bloco de código recebe como parâmetro o objeto de model do FormField(FWFormFieldsModel) e deve retornar um valor lógico.Este bloco é invocado antes da persistência (gravação) dos dados, validando o submodelo.
Se o retorno for verdadeiro, a gravação será realizada se os demais submodelos também estiverem válidos, do contrário um erro será atribuído ao Model, sendo necessário indicar a natureza do erro através da função Help.

- FWFORMMODEL():AddFields([cId], [cOwner], [oModelStruct], [bPre], [**bPost**], [bLoad])-> NIL

Exemplo:

```
Static Function fieldValidPos(oFieldModel)
Local lRet := .T.

   If "@" $ Upper(oFieldModel:GetValue("A1_EMAIL"))
      Help( ,, 'HELP',, 'Informar um emal valido', 1, 0)
      lRet := .F.
   EndIf

Return lRet
```

10.13.3. Estruta Loads do AddFilds

Este bloco será invocado durante a execução do método activate desta classe.
O bloco recebe por parâmetro o objeto de model do FormField(FWFormFieldsModel) e um valor lógico indicando se é uma operação de cópia. Espera-se como retorno um array com os dados que serão carregados no objeto, o array deve ter a estrutura como no exemplo. A ordem dos dados deve seguir exatamente a mesma ordem dos campos da estrutura de dados

- MPFORMMODEL():AddFields(< cId >, < cOwner >, < oModelStruct >, < bPre >, < bPost >, < **bLoad** >)

Exemplo:

```
Static Function loadField(oFieldModel, lCopy)
Local aLoad := {}

   aAdd(aLoad, 1) //recno
   aAdd(aLoad, {xFilial("ZA1"), "000001", "01", "Teste", }) //dados

Return aLoad
```

10.13.4. Validações AddGrid

Um submodelo do tipo Grid permite manipular diversos registros por vez. Ele tem um relacionamento do tipo Nx1 ou NxM com outros Submodelos.

MPFORMMODEL():AddGrid(< cId >, < cOwner >, < oModelStruct >, < bLinePre >, < bLinePost >, < bPre >, < bPost >, < bLoad >)

10.13.5. Estruta Pré Linha Validação do AddGrid

O bloco é invocado na deleção de linha, no undelete da linha e nas tentativas de atribuição de valor.
Recebe como parametro o objeto de modelo do FormGrid(FWFormGridModel), o número da linha atual e a identificação da ação.
A Identificação da ação pode ser um dos itens: "UNDELETE" , "DELETE" , "SETVALUE" nesse caso, serão passados mais três parametros. O 4º parametro é o identificador do campo que está sendo atualizado o 5º parametro é o valor que está sendo atribuido e o 6º parametro é o valor que está atualmente no campo. "CANSETVALUE" : nesse caso será passado mais um parametro. O 4º parametro é o identificador do campo que está tentando ser atualizado.
O retorno do bloco deve ser um valor lógico que indique se a linha está valida para continuar com a ação.
Se retornar verdadeiro, executa a ação do contrário atribui um erro ao Model, sendo necessário indicar a natureza do erro através do método.

MPFORMMODEL():AddGrid(< cId >, < cOwner >, < oModelStruct >, < **bLinePre** >, < bLinePost >, < bPre >, < bPost >, < bLoad >)

Exemplo:

```
Static Function ModelDef()
Local oModel
Local oStruZA1:= FWFormStruct(1,'ZA1')
Local oStruZA2 := FWFormStruct( 1, 'ZA2')
Local bLinePre := {|oGridModel, nLine, cAction, cIDField, xValue, xCurrentValue|
linePreGrid(oGridModel, nLine, cAction, cIDField, xValue, xCurrentValue)}

oModel := MPFormModel():New('COMP021')
oModel:AddFields('ZA1MASTER',,oStruZA1)
oModel:AddGrid( 'ZA2DETAIL', 'ZA1MASTER', oStruZA2, bLinePre, , , ,bLoad)
Return oModel

Static Function linePreGrid(oGridModel, nLine, cAction, cIDField, xValue,
xCurrentValue)
Local lRet := .T.

   If cAction == "SETVALUE"
      If oGridModel:GetValue("ZA2_TIPO") == "AUTOR"
         lRet := .F.
         Help( ,, 'HELP',, 'Não é possível alterar linhas do tipo Autor', 1, 0)
      EndIf
   EndIf

Return lRet
```

10.13.6. Estruta Pós Linha Validação do AddGrid

Código de pós validação da linha do grid, equivale ao "LINHAOK".recebe como parametro o objeto de modelo do FormGrid(FWFormGridModel) e o número da linha que está sendo validada. O bloco será invocado antes da gravação dos dados e na inclusão de uma linha.
Espera-se um retorno lógico do bloco indicando se a linha está ou não valida

MPFORMMODEL():AddGrid(< cId >, < cOwner >, < oModelStruct >, < bLinePre >, < **bLinePost** >, < bPre >, < bPost >, < bLoad >)

10.13.7. Estruta Pré Validação do AddGrid

O bloco é invocado na deleção de linha, no undelete da linha, na inserção de uma linha e nas tentativas de atribuição de valor. Recebe como parametro o objeto de modelo do FormGrid(FWFormGridModel), o número da linha atual e a identificação da ação.
A Identificação da ação pode ser um dos itens "UNDELETE" , "DELETE", "ADDLINE" : nesse caso não será passado nada para o parametro de numero de linha "SETVALUE" nesse caso, serão passados mais três parametros. O 4º parametro é o identificador do campo que está sendo atualizado o 5º parametro é o valor que está sendo atribuido e o 6º parametro é o valor que está atualmente no campo. "CANSETVALUE" : nesse caso será passado mais um parametro. O retorno do bloco deve ser um valor lógico que indique se a linha está valida para continuar com a ação. Se retornar verdadeiro, executa a ação do contrário atribui um erro ao Model

10.13.8. Estrutura Pós Validação do AddGrid

A pós-validação do submodelo, ele é equivalente ao "TUDOOK". O bloco de código recebe como parametro o objeto de model do FormGrid(FWFormGridModel) e deve retornar um valor lógico. Este bloco é invocado antes da persistência(gravação) dos dados, validando o submodelo . Se o retorno do bloco for verdadeiro a gravação será realizada se os demais submodelos também estiverem validos, do contrário um erro será atribuido no Model, sendo necessário indicar a natureza do erro através do modelo.

10.13.9. Estruta Load do AddGrid

Este bloco será invocado durante a execução do método activate desta classe.
O bloco recebe por parametro o objeto de model do FormGrid(FWFormGridModel) e um valor lógico indicando se é uma operação de cópia. Espera-se como retorno um array com os dados que serão carregados no objeto

Exemplo:

```
Static Function ModelDef()
Local oModel
Local oStruZA1:= FWFormStruct(1,'ZA1')
Local oStruZA2 := FWFormStruct( 1, 'ZA2')
Local bLoad := {|oGridModel, lCopy| loadGrid(oGridModel, lCopy)}

oModel := FWFormModel():New('COMP021')
oModel:AddFields('ZA1MASTER',,oStruZA1)
oModel:AddGrid( 'ZA2DETAIL', 'ZA1MASTER', oStruZA2, , , , ,bLoad)
Return oModel

Static Function loadGrid(oModel, lCopy)

Local nI      := 0
Local aFields := oModel:GetStruct():GetFields() // Carrega a estruta criada
Local aDados  := {}
Local aAux    := {}
Local aRet    := {}

 aAux := Array(Len(aFields))

     For ni := 1 to Len(aFields)
          If Alltrim( aFields[ni,3] )== "CODIGO"
                aAux[ni] := "000001"
          ElseIf alltrim(aFields[ni,3]) == "QTD"
                aAux[ni] := 10
          EndIf

     Next nI

     aAdd(aRet,{0 ,aAux })

Return aRet
```

11. Eventos View

11.1. SetCloseOnOk

Metodo que seta um bloco de código para verificar se a janela deve ou não ser fechada após a execução do botão OK na inclusão.

Exemplo:

```
oView:SetCloseOnOk( {|| .T. } )
```

11.2. SetViewAction

Define uma ação a ser executada em determinados pontos da View.

- FWFORMVIEW():SetViewAction(<cActionIID >, <bAction >)

cActionIID: ID do ponto a acao sera 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 acao a ser executada

Exemplo:

```
oView:SetViewAction( 'BUTTONOK' , { |oView| MsgInfo('Apertei OK') } )
oView:SetViewAction( 'BUTTONCANCEL', { |oView| MsgInfo('Apertei Cancelar) } )
oView:SetViewAction( 'DELETELINE', { |oView,cIdView,nNumLine| MsgInfo('Apertei Deletar') } )
```

11.3. SetAfterOkButton

Metodo que seta um bloco de código que será chamado no final da execução do botão OK. O bloco recebe como parametro o objeto de View e não precisa retornar nenhum valor.

- FWFORMVIEW():SetAfterOkButton(<bBlock >)

Exemplo:

```
oView:SetAfterOkButton({|| MsgInfo("Após evento de OK") })
```

11.4. SetViewCanActivate

Metodo que seta um Code-block para ser avaliado antes de se ativar o View, caso precisamos avaliar, ou questionar o usuario sobre algo.No momento de chamada desse bloco o Model e o View não estão ativos, portanto não é possivel

recuperar dados. O bloco recebe como parametro o objeto oView e deve retornar verdadeiro para permitir a ativação. Se retornar falso a janela será fechada.

- FWFORMVIEW():SetViewCanActivate(<bBlock >)

Exemplo:

```
oView:SetViewCanActivate({|| MsgInfo('Pode Ativar'), .F. })
```

11.5. SetAfterViewActivate

Seta um bloco de código que será chamado depois do Activate do View. Esse bloco será apenas executado, o retorno dele não será observado.O bloco de código recebe como parametro o objeto View.

Exemplo:

```
oView:SetAfterViewActivate({|| MsgInfo('Depois de ativado') })
```

11.6. SetVldFolder

Executa ação quando uma aba é selecionada em qualquer folder da View

- oView:SetVldFolder({|cFolderID, nOldSheet, nSelSheet| ValFolder(cFolderID, nOldSheet, nSelSheet)})

Exemplo:

```
Static Function VldFolder(cFolderID, nOldSheet, nSelSheet)
Local lRet := .T.

   If nOldSheet == 1 .And. nSelSheet == 2
      Help( ,, 'Help',, ;
       'Não é permitido selecionar a aba 2 se você estiver na aba 1.', 1,
0 )
      lRet := .F.
   EndIf

Return lRet
```

11.7. SetTimer

Define um timer para a janela do view, o intervalo é em milissegundos , para disparar o bloco de código do Timer.

- FWFORMVIEW():SetTimer(<nInterval >, <bAction >)

Exemplo:

```
oView:SetTimer(10000,{|| MsgInfo('10 Segundo') })
```

12. 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 ideia 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 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.
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 têm 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

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	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
FORMLINEPRE	Antes da alteração da linha do formulário FWFORMGRID. 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 FWFORMGRID 5 C Ação da FWFORMGRID 6 C Id do campo Retorno: Requer um retorno lógico
FORMLINEPOS	Na validação total da linha do formulário FWFORMGRID. 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 FWFORMGRID 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. indica novo registro (Inclusão) se .F. 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. indica novo registro (Inclusão) se .F. 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
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

Exemplo do ponto de entrada em MVC:

```
#Include 'Protheus.ch'

User Function ModelADM()

Local aParam := PARAMIXB

    If aParam <> NIL

          oObj       := aParam[1]
          cIdPonto   := aParam[2]
          cIdModel   := aParam[3]
          lIsGrid    := ( Len( aParam ) > 3 )
             cClasse    := IIf( oObj<> NIL, oObj:ClassName(), '' )

          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 pre 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'
                ApMsgInfo('Antes da gravação da tabela do formulário.
(FORMCOMMITTTSPRE).' + CRLF + 'ID ' + cIdModel)

          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 == 'BUTTONBAR'
```


Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

```
                ApMsgInfo('Adicionando Botao na Barra de Botoes (BUTTONBAR).' +
CRLF + 'ID ' + cIdModel )
                xRet := { {'Salvar', 'SALVAR', { || Alert( 'Salvou' ) }, 'Este
botao Salva' } }
          EndIf

    EndIf

Return xRet
```


Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©

13. Browse com coluna de marcação (FWMarkBrowse)

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

```
FWMarkBrowse

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

// Adiconando botões não necessario ser ter MenuDef
oMark:AddButton("Exemplo Marca", "U_SA2_Marca",  , 1 )

// Definição da tabela a ser utilizada
oMark:SetAlias("SA2")

// Define a titulo do browse de marcacao
oMark:SetDescription( "Cadastro de Grupo" )

// Define o campo que sera utilizado para a marcação
oMark:SetFieldMark( "A2_OK" )

// Ativacao da classe
oMark:Activate()

Return( NIL )

//-------------------------------------------------------------------------
----

User Function SA2_Marca()
Local cMarca := oMark:Mark()

dbSelectArea("SA2")
SA2->(dbGotop())

   while ! SA2->(EOF())
         If oMark:IsMark(cMarca)
               msgInfo(SA2->A2_COD + ' ' + SA2->A2_NOME)
         EndIf

         SA2->( dbSkip() )

   EndDo

   // Executa a atualização das informações no Browse
```

```
   //.T. Indica que deverá ser posicionado no primeiro registro do
Browse
oMark:Refresh(.T.)

Return( NIL )
```

14. Apendice

FWModelActive: Esta função fornece o último objeto da classe FWFormModel ativo, para ser utilizado nas regras de validacao do sistema Microsiga Protheus.
Local oModel := FWModelActive()

FWMemoVirtual(): Alguns campos do tipo MEMO utilizam-se de tabelas para a gravação de seus valores (SYP3), esses campos devem ser informados na estrutura para que o MVC consiga fazer seu tratamento corretamente. 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 abaixo, oStru 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.

oStrut := FWFormStruct(1, "SA1")
FWMemoVirtual(oStrut,{ { 'A1_OBS' } , {'A1_OBS"} })

FWFORMMODELSTRUCT():GetTable() : Fornece os dados da tabela da estrutura.
aRetorno Array com os seguintes dados
- [01] ExpC: Alias da tabela
- [02] ExpA: Array unidimensional com os campos que correspondem a primary key
- [03] ExpC:]Descrição da tabela

FWFORMMODELSTRUCT():GetIndex() : Fornece os dados de todos os índices da estrutura.
aRetorno Array com a definição dos índices
- [01] ExpN: Ordem numérica
- [02] ExpC: Ordem no metadado
- [03] ExpC: Chave do índice
- [04] ExpC: Descrição do índice
- [05] ExpC: LookUp
- [06] ExpC: NickName
- [07] ExpL: Show?

FWFORMMODELSTRUCT():GetFields() : Retorna a coleção de campos da estrutura.
aRetorno Array com a estrutura de metadado dos campos da classe.
- [n] Array com os campos
- [n][01] ExpC: Titulo
- [n][02] ExpC: Tooltip
- [n][03] ExpC: IdField
- [n][04] ExpC: Tipo

[n][05] ExpN: Tamanho
[n][06] ExpN: Decimal
[n][07] ExpB: Valid
[n][08] ExpB: When
[n][09] ExpA: Lista de valores (Combo)
[n][10] ExpL: Obrigatório
[n][11] ExpB: Inicializador padrão
[n][12] ExpL: Campo chave
[n][13] ExpL: Campo atualizavel
[n][14] ExpL: Campo virtual
[n][15] ExpC: Valid do usuario em formato texto e sem alteracao, usado para se criar o aheader de compatibilidade

FWFORMMODELSTRUCT():GetTriggers() : Retorna a coleção de triggers da estrutura.
aRetorno Array com a estrutura de metadado dos triggers da classe.
[n]Array com os triggers
[n][01] ExpC: IdField Origem
[n][02] ExpC: IdField Alvo
[n][03] ExpB: When
[n][04] ExpB: Execução

FWFORMMODELSTRUCT():IsEmpty() : Se verdadeiro, a estrutura está vazia.

FWFORMMODELSTRUCT():HasField(< cIdField >) : Informa de um determinado campo existe na estrutura.

FWFORMMODELSTRUCT():GetFieldPos(<cIdField >) : Retorna a posição na estrutura de um Campo

FWFORMMODELSTRUCT():getLogicTableName() : Retorna o nome fisico da tabela

FWFORMMODELSTRUCT():FieldsLength() : Retorna a quantidade de campos na estrutura.

Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©
Este documento é de propriedade da TOTVS. Todos os direitos reservados. ©