AdvPl utilizando MVC

La arquitectura *Model-View-Controller* o MVC, es conocida como un estándar de Arquitectura de Software que pretende separar la lógica del negocio de la lógica de presentación (Interfaz de Usuario), Lo que permite el desarrollo, pruebas y mantenimiento aislado de ambos.
Aquellos que ya han ~~desarrollo~~ desarrollado una aplicación en ADVPL podrán percibir que justamente la diferencia más importante entre la forma de construir una aplicación en MVC y la forma tradicional, es esa separación y que va a permitir el uso de la regla de negocio en aplicaciones que tengan ~~uno~~ una o más interfaces, como *Web Services* y rutinas automáticas, así como su reuso en otras aplicaciones.

Índice

1. Arquitectura MVC

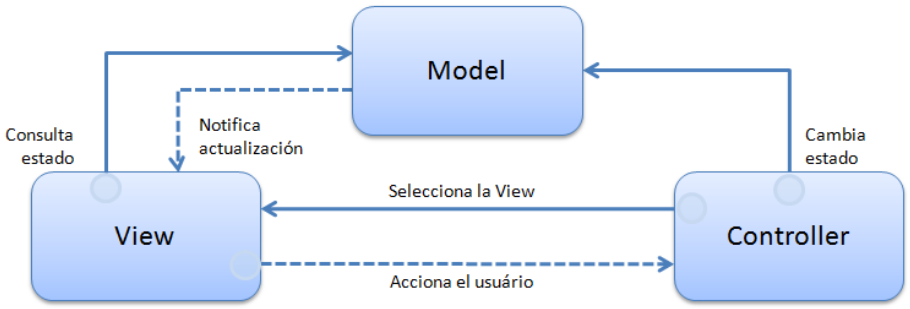
Primero vamos a entender que es la arquitectura *MVC*.

La arquitectura ***Model-View-Controller*** o ***MVC***, es conocida como un ~~patrón~~ estándar de Arquitectura de Software que pretende separar la lógica del negocio de la lógica de presentación (Interfaz de Usuario), ~~Lo~~ lo que permite el desarrollo, pruebas y mantenimiento aislado de ambos.

Aquellos que ya han desarrollado una aplicación en ADVPL podrán percibir que justamente la diferencia más importante entre la forma de construir una aplicación en MVC y la forma tradicional, es esa separación.

~~Y~~ Y es justamente ella que va a permitir el uso de la regla de ~~-~~negocio en aplicaciones que tengan uno o más interfaces, como *Web Services* y aplicaciones automáticas, así como su reuso en otras aplicaciones.

La arquitectura *MVC* tiene tres componentes básicos:



Model o Modelo de Datos: Representa la información del dominio de aplicación y provee funciones para operar los datos, es decir, contiene las funcionalidades de la aplicación. En el definimos las reglas del negocio: Tablas, Campos, Estructuras, Relaciones, etc... El Modelo de Datos (*Model*) también es ~~el~~ responsable por notificar a la Interfaz de Usuario (View) cuando los datos fueron ~~alterados~~modificados.

View o Interfaz de Usuario: Es el responsable por mostrar el modelo de datos (*Model*) y permitir la interacción del usuario, es decir, se encarga de mostrar los datos.

Controller: Responde a las acciones de los usuarios, permite cambios en el ~~Modelo~~ modelo de datos (*Model*) y selecciona la Vista (*View*) correspondiente.

Formatado: Fonte: Itálico

Para ~~-~~facilitar ~~-~~y ~~-~~agilizar ~~-~~el desarrollo, ~~-~~la implementación de ~~-~~MVC ~~-~~se hace en *AdvPL*, ~~-~~el desarrollador trabajará con las definiciones ~~-~~de Modelo de datos ~~-~~(*Model*) ~~-~~y View, la ~~-~~parte responsable por el Controlador (***Controller***) ya se encuentra de forma interna.

Subrayando bien, el gran cambio, el gran paradigma que se rompe en la forma de pensar y de desarrollar una aplicación en *AdvPL* utilizando MVC es la separación de la

regla del negocio, de la Interfaz. Para hacer esto posible se han desarrollado nuevas clases y métodos en *AdvPl*.

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

2. Principales funciones ~~de~~ de la aplicación en AdvPL utilizando MVC

Ahora presentamos el modelo de construcción de una aplicación en AdvPl utilizando MVC.

Los desarrolladores en sus aplicaciones serán los responsables por definir las siguientes funciones:

ModelDef: Contiene la construcción y la definición del Modelo, recordando ~~–~~que el Modelo de datos (*Model*) contiene las reglas del negocio~~;~~.

ViewDef: Contiene la construcción y la definición de la ~~–~~View, es decir, será la construcción de la interfaz~~:~~.

MenuDef: Contiene la definición de las operaciones disponibles para el Modelo de Datos (*Model*).

Cada fuente en *MVC* (PRW) sólo puede contener uno de estos de cada ~~uno~~ una de estas funciones. ~~Solo~~ Sólo puede tener una ***ModelDef***, una ***ViewDef*** y una ***MenuDef***.

Al realizar una aplicación en *AdvPL* utilizando *MVC*, automáticamente al final, esta aplicación tendrá ~~disponible~~a disposición:

- ***Puntos de Entradas*** ya disponibles~~;~~.
- ~~Un~~ Una ***Web Service*** para su utilización~~;~~.
- ***Importación o exportación*** de mensajes *XML*.

~~Podrá ser utilizada,~~Se podrá utilizar de la misma forma que la rutina automática de aplicaciones sin *MVC*.

Un punto importante en la aplicación de MVC es que no se basa necesariamente en metadatos (diccionarios). Como veremos más adelante, se basa en estructuras y estas a su vez pueden venir de metadatos (diccionarios) o ~~serán construidas~~se construirán manualmente.

2.1 ¿Qué es la función ModelDef?

La función ***ModelDef*** define una regla de negocios ~~a propiamente, donde son definidas~~propiamente dicha, donde se definen:

- **Todos los entes** ~~**Todas las entidades**~~ (Tablas) que ~~serán~~ formarán parte del modelo de dados (*Model*~~);~~).
- **Reglas de dependencia** entre ~~las entidades~~los entes~~;~~.
- **Validaciones** (de campos y aplicaciones~~);~~).
- **Persistencia** de datos (Grabación).

Para ***ModelDef*** no necesariamente tiene que haber una interfaz. Como la regla del negocio es totalmente separada de la interfaz en MVC, podemos utilizar la ***ModelDef*** en cualquier otra aplicación, o incluso utilizar una determinada ***ModelDef*** como base para otra más compleja.

Las entidades de ~~—~~***ModelDef*** ~~—~~no se basan necesariamente en metadatos (diccionarios). Como veremos más adelante, se basan en estructuras y estas a su vez pueden venir de metadatos o ser construidas manualmente.

ModelDef debe ser una ***Static Function*** dentro de la aplicación.

2.2 ¿Qué es la función ViewDef?

La función ViewDef define como ~~–~~será la ~~–~~interfaz y, ~~por tanto~~por lo tanto, como el usuario interactúa con el ~~Modelo~~ modelo de datos (*Model*) recibiendo los datos informados por el usuario, ~~proveyendo~~ suministrando ~~al~~ el modelo de datos (definido en ***ModelDef***)y presentando los resultados.

La interfaz ~~puede ser basada~~se puede basar ~~totalmente~~ total o parcialmente en un metadato (diccionario), ~~permitiendo~~que permite:

- **La reutilización** ~~de~~ del código de la interfaz, porque una interfaz básica puede ser ampliada ~~de~~ con nuevos componentes~~;~~.
- **Simplicidad** en el desarrollo de Interfaces complejas. Un ejemplo de esto son aquellas aplicaciones donde un ***GRID*** depende de otro. En ~~—~~MVC la construcción~~—~~ de aplicaciones que tienen ***GRIDs*** dependientes son extremadamente fáciles~~;~~.
- **Agilidad** en el desarrollo, ~~la~~ creación y ~~el~~ mantenimiento. ~~se tornan mucho más ágiles.~~
- **Más de una** interfaz por Bussiness Object. ~~Podremos~~ Podemos tener interfaces diferentes para cada variación de un segmento de mercado, como ~~la Venta al Pormenor~~es el caso del mercado minorista.

La ***ViewDef*** debe ser una ***Static Function*** dentro de la aplicación.

2.3 ¿Qué es la función MenuDef?

Una función MenuDef define las operaciones que serán realizadas por la aplicación, tales como ~~**incluir**~~**Incluir**, ~~**alterar**~~**Modificar**, ~~**excluir**~~**Borrar**, etc.

~~Debe retornar~~En un array ~~en un~~con formato específico debe retornar la siguiente información:

1. Título~~;~~.

2. Nombre de la aplicación ~~asociada~~vinculada~~;~~.

3. Reservado~~;~~.

4. Tipo de ~~Transacción~~ transacción a ~~ser efectuada~~efectuar~~.~~.

~~En la que~~Que pueden ser para:

- 1 ~~para~~ Buscar

- 2 ~~para~~ Visualizar

- 3 ~~para~~ Incluir

- 4 ~~para Alterar~~ Modificar

- 5 ~~para Excluir~~ Borrar

- 6 ~~para~~ Imprimir

- 7 ~~para~~ Copiar

5. Nivel de acceso~~;~~.

6. Habilita Menú Funcional~~;~~.

Ejemplo:

Formatado: Espanhol (Espanha - tradicional)

```
Static Function MenuDef()

Local aRotina := {}

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

Return aRotina
```

Note que el ~~2do~~ 2º parámetro utiliza la llamada ~~directa~~ directamente de una aplicación, en la que se hace referencia a una ViewDef de un determinado fuente (PRW).

La estructura de este ~~2do~~ 2º parámetro tiene el formato:

```
ViewDef.<nombre del fuente>
```

Siempre ~~referenciaremos~~ tomaremos como referencia a la ***ViewDef*** de un fuente, porque es la función responsable por la interfaz de aplicación.

Para facilitar el desarrollo, en *MVC* la función ***MenuDef*** se escribe de la siguiente ~~forma~~manera:

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

```
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
```

El resultado final es el mismo, lo que varia es ~~solo~~ sólo la forma de construcción, pero ~~es más recomendado~~**se recomienda** **la 2da forma** que utiliza el formato de comandos y no las posiciones de un vector, porque ~~en~~ un eventual mantenimiento se ~~tornará~~ volverá más fácil.

MenuDef debe ser una ***Static Function*** dentro de la aplicación.

Utilizando la función ***FWMVCMenu***, se obtiene un menú estándar con las opciones: Visualizar, Incluir, ~~Alterar~~Modificar, ~~Excluir~~Borrar, Imprimir y Copiar. Se debe pasar como parámetro el nombre del fuente.

Por ejemplo:

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

Esto crearía un ***Menudef*** igual que el ejemplo anterior. Para más detalles véase el capítulo **12.7 Creación de un menú estándar (FWMVCMenu).**

2.4 Nuevo comportamiento en la interfaz

Las aplicaciones desarrolladas en *AdvPL* tradicional, después de ~~la finalización de~~finalizar una operación de ~~alteración~~ modificación se cierra la interfaz y ~~retorna~~ vuelve al *Browse*.

Las aplicaciones en *MVC*, después de operaciones de inclusión y ~~alteración~~modificación, la interfaz permanece activa y en la parte inferior se muestra el mensaje de que la operación fue un éxito.

3. Aplicaciones con Browse (FWMBrowse)

Para la construcción de una aplicación que tiene un ***Browse***, *MVC* utiliza la clase ***FWMBrowse***.

Esta clase muestra un objeto ***Browse*** que ~~es construido a partir de~~se crea utilizando metadatos (diccionarios).

Esta clase ***no*** ~~***fue desarrollada***~~***se desarrolló*** ***exclusivamente*** para *MVC*, aplicaciones que no son de MVC también pueden utilizarla. En MVC la utilizaremos.

Sus **característica**s son:

- Sustituir componentes de *Browse*~~;~~ .
- Reducir el tiempo de mantenimiento, en caso de agregar un nuevo requisito~~;~~.
- Ser independiente del ambiente Microsiga Protheus.

Tiene **importantes mejoras**:

- Estandarización de la leyenda de colores~~;~~ .
- Mejor facilidad de uso en ~~-~~el tratamiento de filtros~~;~~ .
- Estandarización de colores, fuentes y leyendas definidas por el usuario - Deficiente visual~~;~~.
- Reducción en el número de operaciones en el SGBD (al menos 3 veces más rápido~~);~~;).
- Nueva estandarización Visual.

3.1 Construcción de un Browse

~~Hablaremos~~ Trataremos aquí de las principales funciones y características para el uso de aplicaciones con MVC.

3.2 Construcción básica de un Browse

Iniciamos la construcción básica de un *Browse*.

Primero hay que crear un objeto Browse de la siguiente forma:

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

Definimos ~~a~~ la tabla que ~~será mostrada~~se mostrará en el ~~-~~Browse utilizando el método ***SetAlias***. Las columnas, órdenes, etc.

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

Definimos el título que aparecerá con el método ***SetDescription***.

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

Y al final activamos la clase.

```
oBrowse:Activate()
```

Con esta estructura básica construimos una aplicación con Browse.

El Browse ~~construido~~ creado automáticamente ~~ya tendrá~~contiene:

- Búsqueda de Registro~~;~~ .
- Filtro configurable~~;~~ .
- Configuración de columnas y apariencia~~;~~.
- Impresión.

3.3 Leyendas de un Browse (AddLegend)

Para el uso de leyendas en un Browse utilizamos un método ***AddLegend***, con la siguiente sintaxis:

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

Ejemplo:

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

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

cRegra es la expresión en AdvPL para definir la leyenda.

cCor es el parámetro que define el color de cada leyenda.

Solo son posibles los siguientes valores:

GREEN	Para el color Verde
RED	Para el color Rojo
YELLOW	Para el color Amarillo
ORANGE	Para el color Naranja
BLUE	Para el color Azul
GRAY	Para el color Gris
BROWN	Para el color Café
BLACK	Para el color Negro
PINK	Para el color Rosa
WHITE	Para el color Blanco

cDescrição Es la que aparece en cada elemento de la leyenda.

Observaciones:

- Cada una de las leyendas se ~~tornará~~ volverá automáticamente una opción de filtro.
- Tenga cuidado al establecer reglas. Si hay reglas en conflicto ~~será mostrada~~se mostrará
  la leyenda correspondiente a la 1ª regla que sea válida.

3.4 Filtros de un Browse (SetFilterDefault)

Si quisiéramos definir un filtro para un *Browse* utilizamos el método ***SetFilterDefault***, que tiene la siguiente sintaxis:

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

Ejemplo:

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

o

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

La expresión del filtro es en ~~-~~AdvPL.

El filtro definido en la aplicación no anula la posibilidad de que el usuario pueda hacer sus propios filtros. Los filtros definidos por el usuario ~~serán aplicados en conjunto~~se aplicarán juntamente con ~~los definidos~~lo definido en la aplicación (Condición de **AND**).

Ejemplo:

Si en la aplicación ~~fue definido que solo muestre los clientes que son~~se definió que se mostrarán sólo las personas jurídicas, y si el usuario hace un filtro para ~~que le muestre~~mostrar solamente los clientes del estado de São Paulo, entonces se ~~mostrará~~ mostrarán los clientes jurídicos del estado de São Paulo. ~~Fue ejecutado el filtro del usuario y sigue siendo respetado~~Se ejecutó el filtro del usuario pero respetando el filtro original de la aplicación.

Observación: El usuario no podrá deshabilitar el filtro de la aplicación~~ no podrá ser deshabilitado por el usuario~~.

3.5 ~~Deshabilitar~~ Cómo deshabilitar~~de~~ los detalles del Browse (DisableDetails)

Automáticamente para el ***Browse*** ~~son mostrados, los detalles, los datos~~se muestran en detalle los datos de la línea posicionada. Para deshabilitar esta característica utilizamos el método ***DisableDetails***.

Ejemplo:

```
oBrowse:DisableDetails()
```

3.6 Campos virtuales en el Browse

~~Normalmente~~Por lo general, para mostrar campos virtuales en los *Browses*, hacemos uso de la función POSICIONE.

En el nuevo *Browse* esta práctica se vuelve aun ~~mas~~ más importante, porque~~,~~ cuando encuentra la función ***Posicione*** definida en un campo virtual y la base de datos es un ***SGBD*** (usa el ***TOTVSDbAccess***), el *Browse* agrega un ***INNER JOIN*** en el query que ~~será enviado~~se enviará ~~-~~al ***SGBD***, mejorando así el desempeño para la extracción de ~~los~~ datos.

~~Por tanto~~Por lo tanto, siempre utilice la función ***POSICIONE*** para mostrar campos virtuales.

3.7 Ejemplo completo de un Browse

Formatado: Espanhol (Espanha - tradicional)

```
User Function COMP011_MVC()
Local oBrowse
// Instanciamento da Classe de Browse
```


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

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

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

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

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

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

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

// Ativação da Classe
oBrowse:Activate()

Return NIL
```

4. ~~Construcción~~ Creación de una aplicación AdvPL utilizando MVC

Iniciamos ahora la ~~construcción~~ creación en MVC de una parte de la aplicación ~~en MVC~~, que son las funciones de ***ModelDef***, la misma que contiene~~n~~ las reglas de negocio, y la ***ViewDef*** que contiene la interfaz.

Un punto ***importante*** que debe ser observado es que, así como la MenuDef, ~~solo~~ sólo **puede haber una función *ModelDef*** y una ***función ViewDef*** en un fuente.

Si para una determinada situación es preciso trabajar en más de un modelo de datos (Model), la aplicación ~~debe ser dividida~~se debe dividir en varios fuentes (PRW) cada uno con ~~apenas~~ solamente una ***ModelDef*** y una ***ViewDef***.

5. ~~Construcción~~ Creación de una aplicación en MVC con ~~una entidad~~un ente

Mostramos como hacer una aplicación en *MVC* con ~~una entidad involucrada~~un ente involucrado.

5.1 Construcción de una estructura de datos (FWFormStruct)

Lo primero que tenemos que hacer es crear la estructura utilizada en el modelo de Datos (Model).

Las estructuras son objetos que contienen las definiciones de los datos necesarios para utilizar el ***ModelDef*** o para el ***ViewDef.***

Estos Objetos Contienen:

- Estructura de los Campos;.
- Índices;.
- GatillosDisparadores;.
- Reglas de llenado (veremos más adelante);).
- Etc.

Como se dijo mencionó anteriormente el MVC no trabaja vinculando vinculado a los metadatos (diccionarios) de Microsiga Protheus, El trabaja vinculandoéste trabaja vinculado a las estructuras. Estas Dichas estructuras, a por su vez, pueden estar construidas a partir dese pueden crear utilizando los metadatos.

Con la función ***FWFormStruct*** la estructura será definidase definirá a partir de metadatos.
La sintaxis es:

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

Donde:

nTipo Tipo de construcción de la estructura:
1 para Modelo de datos (*Model*) y
2 para interfaz (*View*);

cAlias Alias de la tabla en metadatos;

Ejemplo:

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

En el ejemplo, el objeto ***oStruZA0*** será una estructura para usoque se utilizará en un modelo de datos (*Model*). El primer parámetro (1) indica que la estructura es para uso de modelo de datos y el segundo parámetro indica cual la tabla de metadatos será utilizada para la definición dequе se utilizará para definir la estructura (ZA0).

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

En el ejemplo, el objeto ***oStruZA0*** será una estructura para usoque se utilizará en una interfaz (*View*). El primer parámetro (2) indica que la estructura es para definir una interfaz y el segundo parámetro indica la tabla de los metadatos que será utilizada para la definición dese utilizará para definir la estructura (ZA0).

Más adelante veremos cómo definir estructuras manualmente y como seleccionar los campos que formarán parte de las estructuras y otros tratamientos específicos de la estructura.

Importante: Para el modelo de datos (Model), la función ***FWFormStruct*** trae a la estructura todos los campos que componen la tabla, independientemente del nivel, uso o módulo. Considera también los campos virtuales.

Para la interfaz (View) la función ***FWFormStruct,*** trae a la estructura los campos de acuerdo con el nivel, uso u módulo.

5.2 Construcción de la función ModelDef

Como se mencionó anteriormente, en esta función sólo se definen las reglas de negocio o modelo de dados (*Model*).
Ella contiene las definiciones de:

- Entes agregados.
- Validaciones.
- Relaciones.
- Persistencia de datos (grabación).
- Etc.

Iniciamos la función ***ModelDef:***

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

Construcción de la función *Model*:

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

MPFormModel es la clase utilizada para la construcción de un objeto de modelo de dados (*Model*).

Debemos dar un identificador (*ID*) para el modelo como un todo y también uno para cada componente.

Esta es una característica de *MVC*, todo componente de modelo o de interfaz necesita un ID, como formularios, ***GRIDs, boxes***, etc.

COMP011M es un identificador (*ID*) dado al *Model*, y es importante resaltar con relación al identificador (*ID*) del *Model*:

- Si la aplicación es una ***Function***, el identificador (ID) de modelo de datos (*Model*) puede tener el mismo nombre de la función principal y esta práctica se recomienda para facilitar la codificación. Por ejemplo, si estamos escribiendo la función XPTO, el identificador (ID) de modelo de datos (*Model*) podrá ser XPTO.

Formatado: Espanhol (Espanha - tradicional)

- Si la aplicación es una ***User Function*** el identificador (ID) de modelo de datos (Model) **NO** puede tener el mismo nombre de la función principal, esto por causa de los puntos de entrada que son creados automáticamente cuando desarrollamos una aplicación en *MVC*.

Esto ~~será detallado más adelante~~se detallará posteriormente (ver capítulo ***16. Puntos de entrada en MVC***).

5.3 Creación de ~~una~~ un componente de ~~formulários~~ formularios en un modelo de datos (AddFields).

El método ***AddFields*** ~~adiciona~~ agrega un componente de formulario al modelo.

La estructura de modelo de datos (*Model*) debe iniciar, obligatoriamente, con un componente de formulario.

Ejemplo:

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

Debemos dar un identificador (*ID*) para cada componente de modelo.

ZA0MASTER es el identificador (*ID*) dado al componente del formulario en el modelo, ***oStruZA0*** es la estructura que ~~será usada~~ se utilizará en el formulario, ~~el~~ mismo que ~~fue construido~~se construyó ~~anteriormente~~ utilizando ***FWFormStruct***, note que el segundo parámetro (owner) no fue informado, esto porque ~~este~~ es el 1er componente del modelo, el es el ~~padre~~ componente principal ~~de~~ del modelo de ~~dados~~ datos (*Model*) y ~~por tanto~~por lo tanto no tiene un componente superior u ***owner***.

5.4 Descripción de los componentes del modelo de datos (SetDescription)

Siempre ~~definiendo~~ define una descripción para los componentes del modelo.

~~Como~~ Con el método ***SetDescription*** adicionamos la descripción al modelo de datos (*Model*), esa descripción ~~será utilizada~~se utilizará en varios lugares como en *Web Services* por ejemplo.

~~Adicionamos~~ Agregamos la descripción del ***modelo de datos***:

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

~~Adicionamos~~ Agregamos la descripción de los **componentes ~~de~~ del modelo de datos:**

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

Para un modelo que ~~solo~~ sólo contiene un componente parece ser redundante dar una descripción ~~para el~~al ~~modelos~~ modelo de datos *(Model)* como un todo y una ~~para el~~al componente,
pero cuando ~~estudiaremos~~ estudiemos otros modelos donde habrá más de un componente esta acción quedará más clara.

5.5 Finalización de ModelDef

Al final de una función ~~de~~ ***ModelDef***, ~~debe ser retornado~~se debe devolver el objeto de modelo de datos (*Model*) generado en la función.

```
Return oModel
```

5.6 Ejemplo completo del ModelDef

```
Static Function ModelDef()

// Cria a estrutura a ser usada no Modelo de Dados
Local oStruZA0 := FWFormStruct( 1, 'ZA0' )
Local oModel // Modelo de dados que será construído

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

// Adiciona ao modelo um componente de formulário
oModel:AddFields( 'ZA0MASTER', /*cOwner*/, oStruZA0)

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

// Adiciona a descrição do Componente do Modelo de Dados
oModel:GetModel( 'ZA0MASTER' ):SetDescription( 'Dados de Autor/Interprete' )

// Retorna el Modelo de datos
Return oModel
```

5.7 Construcción de la función ViewDef

La ~~interface~~ interfaz (*View*) es responsable por mostrar (hacer, prestar) el modelo de datos (*Model*) y ~~posibilitar~~ permitir la interacción del usuario, es decir, es la responsable por mostrar los datos.

La ***ViewDef*** contiene la definición de toda la parte visual de la aplicación.

Iniciamos la función:

```
Static Function ViewDef()
```

La ~~interface~~ interfaz (View) siempre trabaja basada en un modelo de dados (*Model*). Creamos un objeto de modelo de datos basado en el ***ModelDef*** que deseamos.

Con la función ***FWLoadModel*** obtenemos el modelo de datos (Model) que está definido en un fuente, en ~~nuestro caso es~~este caso se refiere a nuestro propio fuente pero nada nos ~~impediría que utilizáramos~~impide que utilicemos el modelo de cualquier otro fuente en *MVC*, con esto podemos ~~re aprovechar~~reaprovechar un mismo modelo de datos (Model) en más de una interfaz (*View*).~~-~~

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

COMP011_MVC es el nombre del fuente de donde queremos obtener el modelo de datos (*Model*).

Iniciando la construcción de ~~interface~~ la interfaz (View)

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

FWFormView es la clase que ~~deberá ser usada para~~se debe utilizar en la construcción de un objeto de interfaz (*View*).

Definimos cual es el modelo de datos (Model) que ~~será utilizado~~se utilizará en la ~~interface~~ interfaz (View).

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

5.8 Creación de un componente de formularios en la ~~interface~~ interfaz (AddField)

~~Adicionamos~~ Agregamos a nuestra ~~interface~~ interfaz (View) un control ~~de~~ del tipo formulario (antigua ***Enchoice***), para ~~esto~~ ello usamos el método ***AddField***

La ~~interface~~ interfaz (View) se debe iniciar, obligatoriamente, con un componente de tipo formulario.

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

Debemos dar un identificador (*ID*) para cada componente de ~~interface~~ interfaz (*View*).

VIEW_ZA0 es un identificador (*ID*) dado ~~el~~ al componente de ~~interface~~ interfaz (*View*), el oStruZA0 es la estructura que ~~será usada~~se utilizará ~~en~~ y ***ZA0MASTER*** es el identificador (*ID*) de componente del modelo de ~~dados~~ datos (*Model*) vinculado a este componente de la interfaz (*View*).

Cada componente de interfaz (*View*) debe tener un componente de modelo de ~~dados~~ datos (*Model*) relacionado, esto equivale a decir que los datos de ***ZA0MASTER*** ~~serán mostrados~~se mostrarán en la interfaz (*View*) en el componente ***VIEW_ZA0.***

5.9 ~~Exhibición~~ Presentación de los datos en la interfaz (CreateHorizontalBox / CreateVerticalBox)

Siempre necesitamos crear un ***contenedor***[1], un objeto, para recibir algún elemento de la interfaz (*View*). En *MVC* crearemos siempre un ***box*** horizontal ó vertical para ~~esto~~ello.

El método para ~~la creación de~~crear un ***box*** horizontal es:

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

Debemos dar un identificador (*ID*) para cada componente de ~~interface~~ interfaz (View).

TELA (pantalla) es el identificador (*ID*) del ***box*** y el número **100** representa el porcentaje de la pantalla que será utilizado por el Box.

En MVC no hace referencias a coordenadas absolutas de pantalla, los componentes visuales son siempre ***All Client***, es decir, ocuparán todo el ***contenedor*** donde ~~es insertado~~se inserta.

[1] Determinada área definida por el desarrollador para agrupar componentes visuales, por ejemplo, Panel, Dialog, Window, ~~etc~~ etc.

5.10 Relación del componente de interfaz (SetOwnerView)

Necesitamos relacionar el componente de la interfaz (View) con un box para mostrarlo, para esto usamos el método ***SetOwnerView***.

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

De esta ~~forma~~ manera el componente ***VIEW_ZA0*** ~~será mostrado~~se mostrará en la pantalla utilizando el box **TELA**.

5.11 ~~Finalización de~~Cómo finalizar la función ViewDef

Al final de la función ***ViewDef***, ~~debe ser retornado~~se debe devolver el objeto de la interfaz (View) generado.

```
Return oView
```

5.12 Ejemplo completo de ViewDef

```
Static Function ViewDef()
// Crea un objeto de Modelo de datos basado en el ModelDef() del fuente informado
```

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

// Crea la estructura a ser utilizada en el View
Local oStruZA0 := FWFormStruct( 2, 'ZA0' )

// Interfaz de visualización
Local oView
// Crea el objeto del View
oView := FWFormView():New()

// Define cual es el Modelo de datos que será utilizado en la View
oView:SetModel( oModel )
// Adiciona en nuestra View un control de tipo formulário
// (antigua Enchoice)

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

// Crea un "box" horizontal para recibir algún elemento de la view
oView:CreateHorizontalBox( 'TELA' , 100 )

// Relaciona el identificador (ID) de la View como "box"

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

// Retorna el objeto de la View creado


Return oView
```

5.13 Cómo ~~Finalización de~~finalizar la creación de la aplicación ~~como~~ con ~~una entidad~~un ente

De esta forma crearemos una aplicación de *AdvPL* utilizando + donde ~~solo~~ sólo hay una entidad involucrada.

- Construimos el ***ModelDef***~~;~~
- Construimos el ***ViewDef***~~.~~

Esta aplicación sería el equivalente a las aplicaciones ~~de~~ del tipo ***Modelo1*** que normalmente son realizadas.

Más adelante veremos la construcción de aplicaciones utilizando dos o más ~~entidades~~entes.

6. ~~Construcción~~ Cómo crear ~~de~~ una aplicación MVC con dos o más ~~entidades~~entes.

Vimos hasta ahora la ~~construcción~~ creación de una aplicación donde ~~era utilizada solo una entidad~~se utilizaba solamente un ente. Veremos la ~~construcción~~ creación ~~donde~~ en que dos o más ~~entidades~~ entes ~~podrán~~ pueden existir.

La ~~construcción~~ creación de las aplicaciones~~,~~ seguirán los mismos pasos que vimos con anterioridad: ~~Construcción~~ Creación del ***ModelDef*** y de la ~~-~~***ViewDef***. La diferencia básica que hay ahora es que cada una de ellas podrá tener más de un componente y estarán relacionados.

6.1 ~~Construcción~~ Cómo crear ~~de~~ estructuras para una aplicación MVC con dos ó más ~~entidades~~ entes

Como describimos, ~~la primera cosa~~lo primero que debemos hacer es crear la estructura utilizada en el modelo de datos (*Model*). Tenemos que crear una estructura para cada ~~entidad~~ ente que participará en el modelo. Si fueran 2 entidades crearemos 2 estructuras, si fueran 3 entidades crearemos 3 estructuras y así sucesivamente.

Mostraremos una aplicación donde tenemos 2 ~~entidades~~ entes en una relación de dependencia ~~-~~de ***Master-Detail (~~Padre~~ ~~Hijo~~Principal-Secundario)***, como por ejemplo un Pedido de Venta, donde tenemos el encabezado del pedido que ~~sería~~ sería el **Master (~~Padre~~Principal)** y los ítems serían el **Detalle (~~Hijo~~Secundario).**

La ~~construcción~~ creación de las estructuras sería:

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

En el ejemplo anterior el objeto ***oStruZA1***~~,~~ será una estructura ~~para ser utilizada~~que se utilizará en un Modelo de datos (*Model*) para ~~la entidad~~el ente **Master (~~Padre~~Principal)** y ***oStruZA2*** para ~~la entidad~~el ente ***Detalle*** **(~~Hijo~~Secundario).**

El primer parámetro (1) indica que la estructura se ~~va a utilizar~~utilizará en un modelo de datos (Model) y el segundo indica la tabla que se utilizará para ~~construir~~ crear la estructura.

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

En el ejemplo ~~de arriba~~anterior el objeto ***oStruZA1*** será una estructura ~~para ser utilizada~~que se utilizará en una ~~interface~~ interfaz (View) para ~~la entidad~~el ente ***Master (~~Padre~~Principal)*** y ***oStruZA2*** para ~~la entidad~~el ente ***Detalle (~~Hijo~~Secundario).*** El primer parámetro (2) indica que la estructura es para ~~ser~~

~~utilizada~~gue se utilice en una ~~interface~~ interfaz (View) y ~~lo~~ el segundo indica ~~cual~~ la tabla ~~será utilizada para la creación de~~gue se utilizará para crear la estructura.

6.2 ~~Construcción~~ Cómo crear ~~de~~ la función ModelDef

Iniciamos la función ***ModelDef***.

```
Static Function ModelDef()

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

Observe que en ~~el~~ dicho código~~,~~ se crearon 2 estructuras, una para cada ~~entidad~~ente.

Comenzamos ~~la construcción del~~a crear el Model.

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

Debemos dar un identificador (ID) para el Modelo de datos (Model) y para cada componente del Model.

COMP021M es el identificador (ID) dado al Modelo de datos (Model).

6.3 Creación de un componente de formularios en modelo de ~~dados~~ datos (AddFields)

El método ***AddFields*** ~~adiciona~~ agrega al modelo un componente de formulario.

La estructura del modelo debe iniciar, obligatoriamente, con un componente de formulario.

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

Debemos dar un identificador (ID) para cada componente del *Model*.

ZA1MASTER es un identificador (ID) del formulario en el *Model*, ***oStruZA1*** es la estructura que ~~será utilizada~~se utilizará en el formulario y que fue construida anteriormente utilizando ***FWFormStruct***, note que el segundo parámetro (Owner) no fue informado, ~~esto~~ porque este es el 1er componente del *Model*, es el componente ~~padre~~ **principal** del modelo de datos (*Model*) y, ~~por tanto~~por lo tanto no tiene un componente superior u **owner**.

6.4 Creación de un componente de grid en un Modelo de ~~dados~~ datos (AddGrid)

Formatado: Fonte: Itálico

Formatado: Fonte: Itálico

Formatado: Fonte: Itálico

Formatado: Fonte: Itálico

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

La relación de dependencia entre ~~las entidades~~los entes ~~y del~~ es de ***Master-Detail***, es decir, hay 1 ocurrencia del componente ~~**padre**~~ **principal** para "n" ocurrencias del componente ~~hijo~~ secundario (1-n~~)~~).

Cuando ~~una entidad~~un ente ~~-~~ocurra ***n*** veces en el modelo ~~en~~ con relación ~~a otra~~al otro, debemos definir un componente de ***Grid*** para ~~esta entidad~~dicho ente.

El método ***AddGrid*** ~~adiciona~~ agrega ~~en el~~al modelo un componente de *grid*.

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

Debemos dar un identificador (ID) para cada componente del *Model*.

ZA2DETAIL es el ~~-~~identificador (ID) ~~del~~ dado al componente ~~del~~ en el *Model*, ***oStruZA2*** es la estructura que ~~será usada~~se utilizará en el componente y que ~~fue construida~~se elaboró anteriormente utilizando ***FWFormStruct***, note que ~~el segundo parámetro (*Owner*) de esta vez fue informado~~en esta oportunidad el segundo parámetro (Owner) se ha informado, esto ~~es~~ ocurre porque ~~esta~~ dicho ~~entidad~~ente depende de la 1ra (*Master*), ~~por tanto~~por lo tanto ***ZA1MASTER*** es un componente superior ~~u~~ (***owner)*** ~~de~~ a ***ZA2DETAIL***.

6.5 ~~Creación de~~ Cómo crear la relación entre ~~las entidades~~los entes del modelo (SetRelation)

Dentro del modelo debemos relacionar ~~todas las entidades~~a todos los entes que participan ~~de él~~en el mismo. En nuestro ejemplo tenemos que relacionar ~~la entidad~~el ente ***Detail*** con ~~la entidad~~el ente ***Master***.

Una regla ~~de oro simple~~muy sencilla para entender esto es: ~~Toda entidad~~Todo ente ~~de~~ del modelo que tiene un superior (*owner*) debe tener su relación ~~para el hijo~~definida para éste. En otras palabras, es necesario decir cuáles son las ~~llaves~~ claves ~~de relación del hijo para el padre~~de la relación que existe entre los detalles y el principal.

El método que se utiliza para esta definición es ~~-~~***SetRelation***.

Ejemplo:

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

El ***ZA2DETAIL*** es el identificador (ID) ~~de entidad~~del ente ***Detail***, el segundo parámetro es un vector bidimensional donde ~~son definidos~~se definen las relaciones que existen entre cada campo ~~de entidad hijo para la entidad Padre~~del ente secundario y el ente principal. El tercer parámetro es el orden de los datos en el componente.

~~Estamos diciendo~~Lo que queremos decir ~~-~~en el ~~-~~ejemplo anterior, es que ~~-~~la ~~-~~relación ~~de la entidad~~ del ente contable *Detail* ~~será~~se hará ~~-~~por ***ZA2_FILIAL*** ~~-~~y ***ZA2_MUSICA***, ~~-~~el ~~-~~valor ~~-~~de ***ZA2_FILIAL*** ~~será dado~~ se hará por ***xFilial()***~~-~~y el de ***ZA2_MUSICA*** ~~será de~~se hará por ***ZA1_MUSICA***.

Formatado: Fonte: Itálico

Observación: La relación siempre ~~es definido~~se define del ***Detail (~~Hijo~~Secundario)*** ~~para~~ hacia el ***Master (~~Padre~~Principal)***, tanto el identificador (ID) como el orden del vector es ~~bi-dimensional~~bidimensional.

6.6 Definición de ~~llave~~ clave primaria (SetPrimaryKey)

El modelo de datos necesita que siempre se informe cual es la ~~llave~~ clave primaria para ~~la entidad~~el ente principal del modelo de datos (*Model*).

Si la estructura ~~de entidad fue construida~~del ente se construyó utilizando~~-~~ el ***FWFormStruct***, la ~~llave~~ clave primaria será aquella que ~~fue definida~~se definió en los metadados (diccionarios).

Si la estructura fue construida manualmente ó si ~~la entidad~~el ente no tiene definición de ~~llave~~ clave única en metadados, tenemos que definir cuál será esa ~~llave~~ clave con el método ***SetPrimaryKey***.

Ejemplo:

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

~~Donde~~ Aquí, el parámetro ~~pasado~~ que se pasó es un vector con los campos que componen la ~~llave~~ clave primaria**. Use este método ~~solo~~ sólo si es necesario.**

Siempre defina la ~~llave~~ clave primaria para el modelo. Si realmente no fue posible crear una ~~llave~~ clave primaria para ~~la entidad~~el ente principal, informe ~~-~~el modelo de datos de la siguiente forma:

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

6.7 Descripción de los componentes del modelo de datos (SetDescription)

Defina siempre una descripción para los componentes del modelo. Con el método ***SetDescription*** agregamos la descripción del Modelo de ~~Dados~~Datos, esa descripción ~~será utilizada~~se utilizará en varios lugares como en *Web Services* por ejemplo.

Agregamos la descripción del modelo de datos.

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

Agregamos la descripción ~~los~~ de los componentes del modelo de ~~dados~~datos.

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

Note que de esta manera definimos una descripción para el modelo y una para cada componente del modelo.

6.8 Finalización del ModelDef

Al final de la función ***ModelDef***, ~~debe ser retornado~~se debe devolver el objeto del Modelo de datos (Model) generado en la función.

```
Return oModel
```

6.9 Ejemplo completo del ModelDef

```
Static Function ModelDef()

// Crea las estruturas a serán utilizadas en el Modelo de Datos

Local oStruZA1 := FWFormStruct( 1, 'ZA1' )
Local oStruZA2 := FWFormStruct( 1, 'ZA2' )
Local oModel // Modelo de datos construído

// Crea el objeto del Modelo de Datos
oModel := MPFormModel():New( 'COMP021M' )

// Agrega al modelo un componente de formulario
oModel:AddFields( 'ZA1MASTER', /*cOwner*/, oStruZA1 )

// Agrega al modelo un componente de grid
oModel:AddGrid( 'ZA2DETAIL', 'ZA1MASTER', oStruZA2 )

// Hace la relación entre los componentes del model
oModel:SetRelation( 'ZA2DETAIL', { { 'ZA2_FILIAL', 'xFilial( "ZA2" )'
}, { 'ZA2_MUSICA', 'ZA1_MUSICA' } }, ZA2->( IndexKey( 1 ) ) )

// Agrega la descripción del Modelo de Datos
oModel:SetDescription( 'Modelo de Musicas' )

// Agrega la descripción de los Componentes del Modelo de Datos
oModel:GetModel( 'ZA1MASTER' ):SetDescription( 'Dados da Musica' )
oModel:GetModel( 'ZA2DETAIL' ):SetDescription( 'Dados do Autor Da Musica' )

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

6.10 ~~Construcción~~ Creación de la función ViewDef

Iniciamos ~~a~~ la función.

```
Static Function ViewDef()
```

La *interface (View)* siempre trabajará basada en un modelo de datos *(Model).* Creamos un objeto de ~~Modelo~~ modelo de datos basado en ***ModelDef*** que deseamos.

Con la función ***FWLoadModel*** obtenemos el modelo de datos (*Model*) que está definido en un fuente, en nuestro caso, es el propio fuente, ~~mas~~ pero nada impide que utilicemos el modelo de datos (*Model*) de cualquier otro fuente en *MVC*, con esto podemos ~~re aprovechar~~reaprovechar un mismo ~~Modelo~~ modelo de datos (*Model*) en más de una interfaz (*View*).

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

COMP021_MVC es el nombre del fuente de donde queremos obtener el *model*.

Formatado: Fonte: Itálico

Comenzamos la construcción de la ~~interface~~ interfaz (*View*~~)~~).

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

FWFormView es la clase que ~~deberá ser utilizada~~se debe utilizar para ~~la construcción de~~crear un objeto *de interfaz (View).*

Definimos cual es el Modelo de datos (*Model*) que ~~será utilizado~~se utilizará en la interfaz (*View*).

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

6.11 Creación de un componente de formularios en la ~~interface~~ interfaz (AddField)

Agregamos a nuestra interfaz (*View*) un control de tipo formulario (antiguo ***Enchoice***), para esto utilizamos el método ***AddField***.

La interfaz (*View*) debe iniciar, obligatoriamente, con un componente de tipo formulario.

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

Debemos dar un identificador (*ID*) para cada componente de interfaz (*View*). ***VIEW_ZA1*** es el identificador (*ID*) dado al componente de la interfaz (*View*), ***oStruZA1*** es la estructura que ~~será usada~~se utilizará, y el ***ZA1MASTER*** es el identificador (ID) del componente del ~~Modelo~~ modelo de datos (*Model*) vinculado a este componente de la interfaz (*View*).

Cada componente de la interfaz (*View*) debe tener un componente de ~~Modelo~~ modelo de datos (Model) relacionado, esto equivale a decir que los datos del ***ZA1MASTER*** ~~serán mostrados~~se mostrarán en la interfaz (*View*) en el componente ***VIEW_ZA1.***

6.12 Creación de un componente de grid en la interfaz (AddGrid)

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

Agregamos en ~~—~~nuestra interfaz (*View*) un control de tipo grid (antiguo ***GetDados***), para esto utilizamos el método ***AddGrid***.

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

Debemos dar un identificador (*ID*) para cada componente de interfaz (*View*).

VIEW_ZA2 es el identificador (*ID*) dado al componente de interfaz (*View*), ***oStruZA2*** es la estructura que ~~será utilizada~~se utilizará y ***ZA2DETAIL*** es el identificador (*ID*) de componente del ~~Modelo~~ modelo de datos (*Model*) vinculado a este componente de interfaz (*View*).

Cada componente de ~~interface~~ interfaz (*View*) debe tener un componente de ~~Modelo~~ modelo de datos (*Model*) relacionado, esto equivale a decir que los datos de ***ZA2DETAIL*** ~~serán mostrados~~se mostrarán en la interfaz (*View*) en el componente ***VIEW_ZA2***.

Observación: Note que aquí no ~~hablamos~~ comentamos sobre que ~~entidad~~ ente es superior a ~~cual~~otro, esto es porque ~~esta~~ dicha función es del modelo de datos. La interfaz (View) ~~solo~~ sólo refleja los datos del modelo.

6.13 Cómo ~~Mostrar~~ mostrar los datos en la interfaz (CreateHorizontalBox / CreateVerticalBox)

Siempre necesitamos crear un contenedor, un objeto, para recibir algún elemento de la interfaz (*View*).

En MVC crearemos siempre un ***box*** horizontal ~~ó~~ o vertical para esto. El método para ~~la construcción de~~crear un box horizontal es:

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

Debemos dar un identificador (ID) para cada componente de interfaz (*View*). ***SUPERIOR*** es el identificador (ID) del ***box*** y el número ***15*** representa el ~~porcentual~~ porcentaje de la pantalla que ~~será utilizado por~~utilizará el ***box***.

Como tendremos dos componentes necesitamos definir más de un ***box***~~,~~ para el segundo componente.

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

INFERIOR es el identificador (ID) del ***box*** y el número ***85*** representa el ~~porcentual~~ porcentaje de la pantalla que será utilizado por ~~este~~éste.

Observación: La suma de los porcentajes de los dos boxes del mismo nivel deber ser siempre 100%.

6.14 Cómo ~~Relacionar~~ relacionar el componente de la interfaz (SetOwnerView)

Necesitamos relacionar el componente de la interfaz (View) con un box para mostrarlo, para esto utilizamos el método ***SetOwnerView***.

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

De esta ~~-~~forma ~~-~~el ~~-~~componente ***VIEW_ZA1*** ~~-~~será ~~-~~mostrado ~~-~~en la pantalla por el box ~~-~~***SUPERIOR*** ~~-~~y ~~-~~el componente ***VIEW_ZA2*** será mostrado en la pantalla por el box ***INFERIOR***.

Obs.: Note que los datos ~~de entidad padre~~del ente principal ocuparán el 15% de la pantalla y ~~la entidad ***hijo***~~el ente secundario el 85%, porque:

Id del Model	Id del Box	
ZA1MASTER	VIEW_ZA1	SUPERIOR (15%)
ZA2DETAIL	VIEW_ZA2	INFERIOR (85%)

6.15 Finalización del ViewDef

Al final de la función ***ViewDef***, ~~debe ser retornado~~se debe devolver el objeto de la interfaz (View) generado.

```
Return oView
```

6.16 Ejemplo completo de la ViewDef

```
Static Function ViewDef()

// Crea un objeto de Modelo de datos basado en el ModelDef del fuennte informado
 Local oModel := FWLoadModel( 'COMP021_MVC' )

// Crea las estructuras a serán utilizadas en la View
Local oStruZA1 := FWFormStruct( 2, 'ZA1' )
Local oStruZA2 := FWFormStruct( 2, 'ZA2' )

// Interfaz de visualización

Local oView
```

```
// Crea un objeto de View
oView := FWFormView():New()

// Define cual Modelo de dados será utilizado

oView:SetModel( oModel )

// Agrega a nuestra View un control de tipo formulario (antigua Enchoice)
oView:AddField( 'VIEW_ZA1', oStruZA1, 'ZA1MASTER' )

//Adiciona en nuestra View un control do tipo Grid (antiguo Getdados)
oView:AddGrid( 'VIEW_ZA2', oStruZA2, 'ZA2DETAIL' )

// Crea un "box" horizontal para recibir cada
elemento de la view

oView:CreateHorizontalBox( 'SUPERIOR', 15 )
oView:CreateHorizontalBox( 'INFERIOR', 85 )

// Relaciona el identificador (ID) de la View con un "box" para mostrarlo
oView:SetOwnerView( 'VIEW_ZA1', 'SUPERIOR' )
oView:SetOwnerView( 'VIEW_ZA2', 'INFERIOR' )

//  Retorna  el  objeto  de  la  View  creado
Return oView
```

6.17 Cómo ~~Finalización de~~ finalizar la construcción de la aplicación con dos ~~ó~~ o más ~~entidades~~ entes

De esta ~~forma~~ manera construimos una aplicación de *AdvPL* utilizando *MVC* donde hay 2 ~~entidades involucradas~~entes involucrados.

- Construimos ~~la~~ el ***ModelDef***~~,~~
- Construimos ~~la~~ el ***ViewDef***~~.~~

Esta aplicación será el equivalente a las aplicaciones ~~de~~ del tipo ***Modelo3*** que ~~normalmente son construidas~~por lo general se crean.

Si ~~sé necesita la construcción de~~es necesario crear una aplicación con más de 2 ~~entidades~~ entes el proceso será el mismo que el mostrado para 2. La diferencia será solamente la cantidad de cada componente u objeto que ~~serán creados~~se creará.

Para el modelo de datos (*Model*) si la aplicación tiene 3 ~~entidades~~entes, serán necesarias 3 estructuras, 3 componentes ***AddFields*** ~~ó~~ o ***AddGrid*** y 2 relaciones. Si la aplicación tiene 4 ~~entidades~~entes, será necesario 4 estructuras, 4 componentes ***AddFields*** ~~ó~~ o ***AddGrid*** y 3 relaciones, etc.

Para la interfaz (View) si la aplicación tiene 3 ~~entidades~~entes, serán necesarias 3 estructuras, 3 componentes ***AddField*** ~~ó~~ o ***AddGrid*** y 3 boxes. Si la aplicación tiene 4 ~~entidades~~entes, serán necesarios 4 estructuras, 4 componentes ***AddField*** ~~ó~~ o ***AddGrid*** y 4 boxes~~.~~,~~,~~ etc.

El modelo de datos y la interfaz ~~crece~~ crecen en la medida en que ~~crecen~~ crece ~~las~~ la ~~cantidades~~ cantidad de ~~entidades relacionadas~~entes relacionados. Pero la forma básica de construcción es siempre la misma.

7. Tratamientos para el modelo de datos y para la interfaz

Ahora que ya sabemos cómo construir una aplicación en *MVC* utilizando n ~~entidades~~entes, lo que demostraremos en este capítulo son —los tratamientos específicos para algunas necesidades en la construcción de una aplicación para la regla de negocio y para la interfaz, en términos de jerarquía la idea es siempre la misma.

Ejemplo:

- Validaciones~~;~~,
- Permisos~~;~~,
- Movimientos en Líneas~~;~~,
- Obtener y atribuir valores~~;~~,
- Persistencia de los datos~~;~~,
- Crear botones~~;~~,
- Crear folders~~;~~ , etc.

8. Tratamientos para el modelo de dados

Veremos algunos tratamientos que ~~pueden ser realizados~~se pueden llevar a cabo en el modelo de datos (*Model*) conforme a las necesidades:

- Validaciones~~;~~ .
- Comportamientos~~;~~ .
- Manipulación del *Grid*.
- Obtener y atribuir valores en el modelo de dados (*Model*~~);~~).
- Grabación de los -datos manualmente~~;~~ .
- Reglas de llenado.

8.1 Mensajes mostrados en la interfaz

Los mensajes ~~son usados~~se utilizan principalmente durante las validaciones realizadas en el modelo de datos.

~~**Vamos a analizar**~~**Analisemos**: Un ~~ponto~~ punto básico de *MVC* es la separación de la regla de negocio de la interfaz de usuario.
La validación es un proceso ejecutado dentro de la regla de negocio en un eventual mensaje de error que ~~será mostrado~~se mostrará a los usuarios, es un proceso que ~~debe ser ejecutado~~se debe ejecutar en la interfaz, es decir, no puede ser ejecutado en la regla de negocios.

Para ~~trabajar esa~~manejar esta situación ~~fue realizado~~se hizo un tratamiento para la función ***Help***.

La función ***Help*** ~~podrá ser utilizada~~se utilizará en las funciones dentro del modelo de datos (*Model*), pero el MVC ~~irá a guardar~~guardará ~~ese~~ este mensaje y ~~esta~~ ésta ~~solo~~ sólo ~~será mostrado~~se mostrará cuando el control vuelva ~~para~~ a la interfaz.

Por ejemplo, una determinada función contendrá:

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

Suponiendo que el mensaje de error ~~fue detonado~~se activó porque el precio unitario es 0 (cero), en este momento no ~~será mostrado~~se mostrará nada al usuario, esto ~~puede ser observado~~se puede ver al ~~debuguear~~ hacer debug en el fuente. Verá que al pasar por la función ***Help*** nada ~~acontece~~sucede, pero~~,~~ cuando el control interno ~~vuelva~~ vuelve a la interfaz, ~~el mensaje es mostrado~~aparece el mensaje.

Este tratamiento ~~fue realizado solo~~ sólo se realiza para la función ~~-~~***Help***, funciones como ~~-~~***MsgStop***, ~~-~~***MsgInfo***, ***MsgYesNo***, ***Alert***, ***MostraErro***, etc. ***No~~ podrán~~*** ~~ser utilizadas~~se utilizarán.

8.2 ~~Obtener~~ Cómo obtener el componente del modelo de datos (GetModel)

Durante el desarrollo ~~varias veces tendremos que manipular~~tenemos que utilizar varias veces el modelo de datos (Model), para facilitar ~~esa manipulación~~este uso podemos ~~en lugar de trabajar con todo el modelo,~~ trabajar ~~solo~~ sólo con una parte específica (un componente)~~, en lugar de trabajar con todo el modelo~~.

Para ~~esto~~ ello utilizamos el método ***GetModel***.

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

oModelZA2 es el objeto que contiene un componente del modelo de datos (*Model*) y ***ZA2DETAIL*** es el identificador (ID) del componente que queremos.

Se ~~tuviéramos~~ tenemos una parte del modelo de datos (*Model*) y ~~quisiéramos pegar~~queremos adoptar el modelo completo también podemos usar el ***GetModel***.

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

oModel es el objeto que contiene el modelo de datos (*Model*) completo.

8.3 Validaciones

Dentro del modelo de datos ~~existentes hay varios puntos donde pueden ser insertadas las validaciones necesarias~~existen varios puntos que permiten insertar las validaciones necesarias para ~~a~~ la regla del negocio. El modelo de datos (*Model*) ~~-~~como un todo tiene sus propios puntos y cada componente ~~de~~ del modelo~~, también sus puntos~~también los tiene.

8.3.1 ~~Pos-validación~~Postvalidación del modelo

Es la validación realizada después de informar el modelo de datos (*Model*) y ~~de~~ su confirmación. Sería el equivalente al antiguo proceso de ***TudoOk***.

El modelo de datos (*Model*) ~~ya~~ hace la validación ~~de~~ si los campos obligatorios de todos los componentes del modelo ~~que fueron informados~~se han completado, ~~esa~~ esta validación ~~es ejecutada~~se ejecuta después de esto.

Definimos la ~~pos-validación~~postvalidación ~~-~~del ~~-~~modelo de ~~-~~datos ~~-~~(*Model*) ~~-~~como ~~-~~un bloque de ~~-~~código en el ~~-~~3er parámetro de la clase de construcción del modelo ***MPFormModel***.

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

Un bloque de código recibe como parámetro un objeto que es el modelo y ~~que~~ se puede ~~ser pasado~~pasar a la función que hará la validación.

```
Static Function COMP011POS( oModel )

Local lRet := .T.
Local nOperation := oModel:GetOperation
      // Sigue la función ...
Return lRet
```

La función llamada por el bloque de código debe retornar un valor lógico, ~~donde~~ si es .T. (verdadero) la operación ~~es realizada~~se realiza, si es .F. (falso) la operación no se realiza.

8.3.2 ~~Pos-validación~~Postvalidación de línea

En un modelo de datos (Model) donde existen componentes de grid, ~~puede ser definida~~se puede definir una validación que ~~será ejecutada~~ se ejecuta en el cambio de líneas del grid. Sería el equivalente ~~en el~~ al antiguo proceso de ***LinhaOk***.

Definimos la ~~pos-validación~~postvalidación de línea como un bloque de código en el 5to parámetro del método ***AddGrid***.

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

El bloque de código recibe como parámetro un objeto que es la parte del modelo correspondiente al grid y que ~~puede ser pasado para~~se puede pasar a la función que hará la validación.
La función llamada por el bloque de código debe retornar un valor lógico, donde si .T. (verdadero) cambia de línea ~~y~~ significa que ~~es realizada~~se realiza la operación y si es~~-~~ .F. (falso) no se realiza el cambio de línea.

8.3.3 Validación de línea duplicada (SetUniqueLine)

En un modelo de datos donde existen componentes de *grid* ~~pueden ser definidos~~se definen cuales son los ~~-~~campos que no se ~~pueden~~ deben repetir dentro de este *grid*.

Por ejemplo, imaginemos ~~el~~ un Pedido de Ventas ~~y no podemos~~en el que no debemos permitir que el código de producto se repita, ~~podemos definir en un modelo este comportamiento~~lo que debemos hacer es definir este comportamiento en un modelo, sin la necesidad de escribir ninguna función específica para ~~esto~~ello.

El método del modelo de datos (Model) que ~~debe ser utilizado~~se debe utilizar es ***SetUniqueLine***.

```
// Liga o control de no repetición de la línea
oModel:GetModel( 'ZA2DETAIL' ):SetUniqueLine( { 'ZA2_AUTOR' } )
```

En el ejemplo anterior ~~el~~ el contenido del campo ***ZA2_AUTOR*** no ~~podrá tener su contenido repetido~~no se debe repetir en el grid.

También ~~puede ser informado~~se puede informar más de un campo, creando así un control con ~~llave~~ clave compuesta.

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

En el ejemplo anterior la combinación de un campo ***ZA2_AUTOR*** y ***ZA2_DATA*** no ~~pueden~~ debe tener su contenido repetido en el *grid*.

Individualmente la repetición ~~-~~podrá ocurrir, ~~más~~ pero no en conjunto.

ZA2_AUTOR	ZA2_DATA	
001	01/01/11	Ok
001	02/01/11	Ok
002	02/01/11	Ok
001	01/01/11	No permitido

8.3.4 ~~Pre-validación~~Prevalidación de ~~la~~ línea

En un modelo de datos donde existen componentes de *grid* ~~puede ser definida~~se puede definir una validación que ~~será ejecutada~~se ejecutará en ~~mas~~ las ~~acciones~~ actividades de las líneas del *grid*. Podemos entender por estas ~~acciones~~ actividades la ~~asignación~~ atribución de valores, ~~apagar~~ borrar o recuperar una línea.

Definimos la ~~pre-validación~~prevalidación de línea como un bloque de código en el 4to parámetro del método ***AddGrid***.

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

El bloque de código recibe como parámetros:

- Un objeto que es a parte del modelo correspondiente al *grid*~~;~~.
- El número de línea~~;~~.
- La acción ejecutada~~;~~.
    - ***SETVALUE*** - Para la asignación de valores~~;~~.
    - ***DELETE*** - Para eliminación y recuperación de la línea.

El campo ~~donde se está asignando~~al que se está atribuyendo el valor, para ~~declaración y recuperación de~~borrar y recuperar la línea no ~~es pasado~~se transfiere.
~~Esos~~ Dichos parámetros ~~pueden ser pasados para~~se deben pasar a la función que ~~hará la~~se encargará de la validación.

La función llamada por el bloque de código debe retornar un valor lógico, ~~donde~~ si es .T. (verdadero) cambia de línea y si es .F. (falso) no la cambia.

Un ejemplo de ~~utilización de~~cómo utilizar la ~~pre-validación~~prevalidación de línea:

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

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

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

Return lRet
```

En el ejemplo anterior no ~~será permitida~~se permitirá la eliminación de la línea en la operación de ~~modificar~~modificación.

8.3.5 ~~Validación de~~Cómo validar la activación del modelo (SetVldActivate)

~~En~~ Es la validación ~~realizada~~ que se realiza ~~en el momento de la activación del~~cuando se activa el modelo, permitiendo o no su activación. Definimos la validación de la activación usando o método ***SetVldActivate***.

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

El bloque de código recibe como parámetro un objeto que es del modelo correspondiente, pero~~,~~ el modelo todavía no tiene los datos cargados, porque la carga de los datos ~~es realizada después de su activación~~se hace después de activarlo.~~-~~

La función llamada por el ~~-~~bloque de código debe retornar un valor lógico, ~~donde~~ si es .T. (verdadero) la activación ~~es realizada~~se realiza y si es .F. (falso) no ~~es realizada~~se realiza.

8.4 ~~Manipulación~~ Manejo del componente grid

Veremos ahora algunos tratamientos que ~~pueden ser realizados~~se pueden realizar en los componentes de *grid* de un modelo de datos (*Model*).

8.4.1 Cantidad de líneas del componente grid (Length)

Para obtener el número de líneas del *grid* debemos utilizar el método ***Length***. Las líneas ~~apagadas~~ borradas también ~~son consideradas en el conteo~~se consideran en el cómputo.

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

For nI := 1 To    oModelZA2:Length()
    // Sigue la Función ...
Next nI
```

Si ~~pasas~~ un parámetro ~~al~~ se pasa por el método ***Length***, el retorno será tan sólo el número de líneas no ~~apagadas~~ borradas en el grid.

```
nLinhas := oModelZA2:Length( .T. ) // Cantidad de líneas activas
```

8.4.2 Ir para una línea del componente grid (GoLine)

Para ~~movernos en~~mover el ~~*grid*~~, ~~es decir~~, ~~cambiarnos de~~cambiar la línea~~, de~~ donde el *grid* ~~está~~ posicionado, utilizamos el método ***GoLine***, ~~pasando~~ y pasamos como parámetro el número de la línea ~~en~~ donde se desea posicionar.

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

For  nI  :=  1  To  oModelZA2:Length()
      oModelZA2:GoLine( nI )
      //  Sigue  la  función  ...
Next nI
```

8.4.3 ~~Status~~ Estatus de la línea de un componente de grid

Cuando ~~estamos hablando~~hablamos del modelo de datos (*Model*) tenemos 3 operaciones básicas: ***Incluir***, ***Modificar*** y ***Borrar***.

Formatado: Fonte: Itálico

Cuando la operación es de inclusión, todos los componentes del modelo de datos (*Model*) están ~~en estatus de inclusión~~incluidos, ~~ese~~ este razonamiento también se aplica ~~a exclusión~~al borrado, si esta es la operación, ~~todos los componentes tendrán sus datos en estatus de exclusión~~se borrarán los datos de todos los componentes.~~.~~

Formatado: Fonte: Itálico

Pero, cuando hablamos de la operación de ~~alteración~~modificación, no es exactamente así.

En un modelo de dados donde existan componentes de *grid*, la operación ~~de alteración~~que permite modificar ~~del~~ el *grid* puede tener líneas incluidas, ~~alteradas ó~~modificadas o ~~excluidas~~borradas, es decir, el modelo de datos (*Model*) está en ~~alteración~~ modificación, pero ~~el~~ un *grid* puede ~~tener estatus en~~haber tenido las 3 operaciones en sus líneas.

En *MVC* es posible saber que operaciones ~~de~~ ha sufrido una línea ~~han sufrido por~~ en los siguientes métodos de ~~status~~estatus:

IsDeleted: Informa si una línea ~~fue apagada~~se ha borrado. ~~Retornando~~Si vuelve .T. (verdadero) indica que la línea ~~fue apagada~~se borró.

IsUpdated: Informa si una línea ~~fue alteradase ha modificado~~. ~~Retornando~~ Si vuelve .T. (verdadero) indica que la línea ~~fue alteradase modificó.~~

IsInserted: Informa si una línea ~~fue incluidase ha incluido~~, es decir, si es una línea nueva en un grid. ~~Retornando~~ Si vuelve .T. (verdadero) indica que la línea ~~fue insertadase incluyó~~.

Ejemplo:

```
Static Function COMP23ACAO()

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

Local nCtInc       := 0
Local nCtAlt       := 0
Local nCtDel       := 0
Local aSaveLines   := FWSaveRows()

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

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

Next

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

Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)

Más de un método de ~~status~~ estatus puede ~~retornar~~ volver .T. (verdadero) para la misma línea. Si una línea ~~fue incluida~~se ha incluido, la ***IsInserted*** ~~retornará~~ volverá .T. (verdadero), si ~~fue alterada~~se ha modificado, la ***IsUpdated*** ~~retornará~~ volverá .T. (verdadero), si la misma línea ~~fue apagada~~se ha borrado, ***IsDeleted*** también ~~retornará~~ volverá .T. (verdadero).

8.4.4 ~~Adicionar~~ Cómo agregar una línea al grid (AddLine)

Para agregar una línea a un componente de grid del modelo de datos (*Model*) utilizamos el método ***AddLine***.

Formatado: Fonte: Itálico

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

El método ***AddLine*** ~~retorna~~ devuelve la cantidad ~~de~~ total de líneas del *grid*. Si el *grid* ya tiene 2 líneas y ~~todo~~ lo demás ha ~~ido~~ resultado bien ~~en la adición de~~al agregar la línea, el ~~-~~***AddLine*** devolverá 3, si ha ocurrido un problema devolverá 2, ya que la nueva ~~fila~~ línea no se ha insertado.

Los motivos para que la inclusión no ~~sea correcta podrán ser que~~haya tenido éxito, debe ser por que algún campo obligatorio no se ha informado, la ~~pos-validación~~postvalidación de la línea ~~retorno~~ volvió .F. (falso), ~~alcanzo~~ alcanzó la cantidad máxima de líneas para el *grid*, por ~~mencionar algunas~~ejemplo.

8.4.5 ~~Apagar~~ Cómo borrar y recuperar una línea de grid (DeleteLine y UnDeleteLine)

Para ~~apagar~~ borrar una línea de un componente de grid del modelo de datos (*Model*) utilizamos el método ***DeleteLine***.

Formatado: Fonte: Itálico

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

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

Next
```

El método ***DeleteLine*** ~~retorna~~vuelve .T. (verdadero) si ~~el apagado de la línea fue correcto~~la línea se ha borrado correctamente. Un motivo para que no sea correcto es que la ~~pre-validación~~prevalidación de la línea ~~retorne~~ vuelva .F. (falso).

Si ~~quisiéramos~~ queremos recuperar una línea de un grid que está ~~apagada~~ borrada utilizamos el método ***UnDeleteLine***.

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

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

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

Next
```

El método ***UnDeleteLine*** retorna .T. (verdadero) si la recuperación fue realizada con éxito. Un motivo para que no lo sea es que la ~~pre validación~~prevalidación de la línea retorne .F. (falso).

8.4.6 Permisos para un grid

Si quisiéramos limitar para que una línea de un grid se pueda ~~ser insertada~~incluir, ~~modificada ó apagada~~modificar o borrar, para hacer una consulta, por ejemplo, utilizamos uno de los métodos abajo:

SetNoInsertLine: No permite que se ~~inserten~~ incluyan líneas en el grid.

Ejemplo:

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

SetNoUpdateLine: No permite que se ~~alteren~~ modifiquen las líneas del grid.

Ejemplo:

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

SetNoDeleteLine: No permite que se ~~apaguen~~ borren líneas ~~del~~ en el grid.

Ejemplo:

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

~~Esos~~ Estos métodos ~~pueden ser informados~~se pueden informar en el momento de la definición del modelo de datos (*Model*).

8.4.7 Permiso de grid vacio (SetOptional)

Por estándar, cuando tenemos un modelo de datos (*Model*) donde hay un componente de *grid*, ~~debe ser informada~~se debe informar por lo menos una línea en este *grid*.

Más vamos a imaginarPero imaginemos un modelo donde tengamos un cadastro archivo de productos y sus componentes. En un modelo *Master-Detail*, tendremos para cada producto ***n*** componentes, pero también tendremos productos que no tengan componente algunoningún componente. AsíPor lo tanto, que esta regla de que, debe haber por lo menos una línea informada en un *grid* no puede ser aplicadase debe aplicar.

En este caso utilizamos el método ***SetOptional*** para permitir que el *grid* tenga ó o no por lo menos una línea digitada, es decir, para decir considerar que la introducción digitación de datos del *grid* es opcional.

Este método debe ser informadose debe informar al definir el modelo de datos (*Model*).

Ejemplo:

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

Si un *grid* es opcional y en una estructura hay campos obligatorios, solo solamente será validadose validará si estos campos fueran se han informados y solamente tan sólo si la línea sufre alguna alteración modificación en cualquier campo.

El método ***IsOptional*** puede ser utilizadose puede utilizar para saber si un componente de *grid* tiene ó o no esta característica. Se retornaSi vuelve .T. (verdadero) si el componente permite que no existan líneas digitadas. Este método pode ser útil en algunas validaciones.

8.4.8 Guardando y restaurandoCómo guardar y restaurar el posicionamiento de un grid (FWSaveRows / FWRestRows)

Un cuidado que debemos tener cuando escribimos una función, mismo queaunque no sea para uso en *MVC*, es restaurar las áreas de las tablas que des posicionamosdescolocamos.
Del mismo modo, debemos tener el mismo cuidado para los componentes del *grid* que des posicionamosdescolocamos en una función, como el uso del método ***GoLine***, por ejemplo.

Para esto utilizaremos las funciones ***FWSaveRows*** para a fin de guardar el posicionamiento de las líneas de los *grids* del modelo de datos (*Model*) y el ***FWRestRows*** para restaurar esos dichos posicionamientos.

Ejemplo:

```
Static Function COMP23ACAO()

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

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

```
            oModelZA2:GoLine( nI )

            // Sigue la Función….
        Next

        FWRestRows( aSaveLine )
```

Obs.: El ***FWSaveRows*** guarda la posición de todos los *grids* del modelo de datos (*Model*) y el ***FWSaveRows*** restaura el posicionamiento de todos los *grids* del model.

8.4.9 Definición de la cantidad máxima de líneas del grid (SetMaxLine)

Por estándar, la cantidad máxima de líneas de un componente del grid es de 990.

Si es necesario modificar esta cantidad se utiliza el método ***SetMaxLine***. Este método debe ser usado en la definición del modelo de datos (*Model*), es decir, en la ***ModelDef***.

Importante: La cantidad se refiere siempre al total de líneas, independientemente si estas están ~~apagadas ó~~borradas o no.

8.5 ~~Obtener y asignar~~Obtención y atribución de valores ~~al~~ en el modelo de datos

Las operaciones más comunes que haremos en un modelo de datos (*Model*) es obtener y ~~asignar~~ atribuir valores.

Para ~~esto~~ ello utilizamos uno de los métodos:

GetValue: Obtiene un dato del modelo de datos (Model). Podemos obtener el dato a partir del modelo completo ~~ó~~ o a partir de su componente.

A partir del modelo de datos (*Model*) completo.

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

Donde ***ZA1MASTER*** es el identificador (ID) del componente y es ***ZA1_MUSICA*** es el campo del cual se desea obtener el dato.

O a partir de un componente de modelo de dados (*Model*).

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

SetValue: ~~Asignar~~ Atribuir un dato del modelo de dados (*Model*). Podemos ~~asignar~~ atribuir un dato a partir del modelo completo ~~ó~~ o a partir de algún componente.

A partir del modelo de datos (*Model*) completo

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

Donde ***ZA1MASTER*** es el identificador (ID) del componente y ***ZA1_MUSICA*** es el campo en el cual se desea ~~asignar~~ atribuir el dato, y *000001* es el dato que se desea ~~asignar~~atribuir.

O a partir de un componente del modelo de ~~dados~~ datos (*Model*).

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

Cuando utilizamos ~~-~~***SetValue*** para asignar un dato a un campo, las validaciones de este campo ~~son ejecutadas y también son disparados sus gatillos~~se ejecutan y también se disparan sus gatillos.

El ***SetValue*** retorna .T. (verdadero) si la ~~asignación fue bien realizada~~atribución tuvo éxito, los motivos para ~~que~~ no ~~pueda ser asignado~~atribuir el dato con éxito, es ~~que~~ cuando ~~el~~ dicho dato no ~~satisfaga~~ cumple con la validación ~~ó~~ o el modo de edición (***WHEN***) ~~no fue satisfecho~~no se ha cumplido, etc.

LoadValue: Cargar un dato del modelo de dados (*Model*). Podemos ~~asignar~~ atribuir el dato a partir del modelo completo ~~ó~~ o a partir de una parte de ~~él~~este.

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

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

Donde ***ZA1MASTER*** es el identificador (ID) del componente y ***ZA1_MUSICA*** es el campo donde se desea ~~asignar~~ atribuir el dato, y ***000001*** es el dato que se desea ~~asignar~~atribuir.

~~Ó~~ O a partir de un componente del modelo de datos (*Model*).

```
Local oModelZA2 := oModel:GetModel( 'ZA1MASTER' )


oModelZA2:LoadValue('ZA1_MUSICA', '000001' )
```

A diferencia entre el ***LoadValue*** y el ***SetValue***, es que el ***LoadValue*** no ejecuta las validaciones ni dispara los gatillos del campo, esta forza la ~~asignación~~ atribución del dato.

Importante: Utilice siempre el ***SetValue*** para ~~asignar~~ atribuir un dato, evite el LoadValue, solo utilícelo cuando es extremadamente necesario.

Formatado: Inglês (Estados Unidos)

Formatado: Inglês (Estados Unidos)

Formatado: Inglês (Estados Unidos)

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

8.6 Comportamiento

Veremos cómo modificar algunos de los comportamientos estándares del modelo de datos (*Model*).

8.6.1 No Modificar los datos de un componente del modelo de datos (SetOnlyView)

Si quisiéramos que un determinado componente de modelo de datos (*Model*) no permita ~~alteración en~~modificar ~~—~~sus datos, y que sea solo para ~~visualización~~visualizar, utilizamos el método ***SetOnlyView***.

Este método debe ser informado en el momento de la definición del Model.

Ejemplo:

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

8.6.2 No grabar datos de un componente del modelo de datos (SetOnlyQuery)

La grabación de los datos ~~es hecha~~se hace automáticamente por el modelo de datos (Model).

Si quisiéramos que un determinado componente de modelo de datos (Model) permita ~~inclusión~~ incluir y/~~ó~~ o ~~alteración~~ modificar ~~en~~ sus datos, pero~~,~~ que estos datos no sean grabados, utilizaremos el método ***SetOnlyQuery***.

Este método debe ser informado en el momento de la definición del Model.

Ejemplo:

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

8.6.3 Obtención de la operación que ~~está siendo realizada~~se está realizando (GetOperation)

Para saber la operación con que un modelo de datos (*Model*) está trabajando, utilizamos el método ***GetOperation***.

Este método retorna:

- El valor 3 cuando es una ***inclusión~~;~~***
- El valor 4 cuando es una ***~~alteración~~;modificación***
- El valor 5 cuando es una ***~~exclusión~~.borrado***

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

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

// Valida si puede o no apagar una línea del Grid
If cAcao == 'DELETE' .AND. nOperation == 3

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

Return lRet
```

En MVC fueron creadas varias directivas de compilación ***#DEFINE*** para facilitar el Desarrollo y regresar a lectura de una aplicación más fácil.

Para utilizar este ***#DEFINE*** es necesario incluir la siguiente directiva en el fuente:

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

Para las operaciones del modelo de datos (*Model*) pueden ser utilizados:

- ***MODEL_OPERATION_INSERT*** para **inclusión~~;~~**
- ***MODEL_OPERATION_UPDATE*** para **~~alteración;~~modificación**
- ***MODEL_OPERATION_DELETE*** para **~~exclusión.~~borrado**

Así mismo, en el ejemplo podemos escribir:

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

8.6.4 Grabación manual de datos (FWFormCommit)

La grabación de los datos del modelo de datos (*Model*) es realizada por MVC donde son grabados todos los datos de ~~las entidades~~los entes del *model*.

Pero, puede haber la necesidad de efectuar grabaciones en ~~otras entidades~~otros entes –que –no participan del modelo. Por ejemplo, cuando incluimos un Pedido de Ventas es necesario actualizar el valor de pedidos en el ~~Cadastro~~ Archivo de Clientes. El encabezado e ítems del pedido hacen parte del modelo y serán grabados, el cadastro de Cliente no hace parte del modelo, pero es necesario que también sea actualizado.

Para este tipo de situación es posible intervenir el momento de la grabación de los datos.

Para esto definimos un bloque de código, en el 4to parámetro de la clase de construcción del modelo de datos (*Model*) ***MPFormModel***.

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

El bloque de código recibe como parámetro un objeto que es el modelo y que puede ser pasado a la función que hará la grabación.

A diferencia de los bloques de código definidos en el modelo de datos (*Model*) para validación, que complementan las validaciones hechas por el *MVC*, el bloque de código para grabación reemplaza la grabación de los dados. Entonces al ser definido un bloque de código para grabación, pasa ser responsabilidad de la función creada, la grabación de todos los datos ~~inclusive~~ incluso los dados del modelo de datos en uso.

Para facilitar el desarrollo fue creada la función ***FWFormCommit*** que hará la grabación de los datos del objeto del modelo de datos (*Model*) informado.

```
Static Function COMP011GRV ( oModel )

FWFormCommit( oModel )

// Efectuar la grabación de otros dados en entidades que
// no son del model
```

Importante: No ~~deben ser hechas asignaciones~~se deben hacer atribuciones de datos al modelo (*Model*) dentro de la función de grabación. Conceptualmente al iniciar la grabación, el modelo de datos (*Model*) ya ~~paso~~ pasó por toda la validación, al intentar ~~asignar~~ atribuir un valor, este valor puede no satisfacer la validación del campo, haciendo que el modelo de datos (*Model*) ~~invalidado~~ se invalide nuevamente y lo que ocurrirá es la grabación de datos inconsistentes.

8.7 Reglas ~~de llenado~~para completar (AddRules)

Una nueva característica que ~~fue implementada~~se implementó en MVC son las reglas ~~de llenado~~para completar, donde ~~el llenado de un campo depende del llenado~~completar um campo depende de cómo se complete ~~de~~ el otro.

Por ejemplo, podemos definir que en el campo Código de Loja de ~~una entidades~~un ente, solo puede ser informado después ~~del llenado del~~de completar el campo Código ~~de~~ del Cliente.

Las reglas ~~de llenado~~para completar pueden ser de 3 tipos:

- **Tipo 1 ~~Pre-Validación~~Prevalidación**

Agrega una relación de dependencia entre campos del formulario, impidiendo la ~~asignación~~ atribución de valor en caso de que los campos de dependencia no tengan valor ~~asignado~~atribuido. Por ejemplo, ~~el llenado de~~para completar un campo Código de Loja solo se puede ~~ser llenado~~realizar después ~~del llenado del~~de completar el campo Código de Cliente.

- **Tipo 2 Pos-Validación**

Agrega una relación de dependencia entre la referencia de origen y destino, provocando una revalidación de destino en caso de actualización del origen. Por ejemplo, después ~~del llenado de un~~de completar el campo Código de ~~Loja~~ la Tienda, ~~la~~

~~validación en caso de que el Código del Cliente sea alterado~~se vuelve a evaluar la validación si el código del cliente se modifica.

- **Tipo 3 Pre y ~~Pos Validación~~Postvalidación**

Son los tipos 1 y 2 simultáneamente.

Ejemplo:

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

~~Donde el~~El ***ZA3MASTER*** es el identificador (ID) del componente del modelo de datos (*Model*) donde está el campo de destino, el ***ZA3_LOJA*** es el campo destino, el segundo ***ZA3MASTER*** es del componente del modelo de datos (*Model*) donde está el campo de origen, y ***ZA3_DATA*** ~~-~~es el campo de origen.

9. Tratamientos de la interfaz

Veremos algunos tratamientos que pueden ser realizados en la interfaz (View) ~~conforme a~~según las necesidades.

- Creación de botones~~;~~
- Creación de folders~~;~~
- ~~Agrupamiento~~ Agrupación de campos~~;~~
- Incremento de campos~~;~~
- Etc.

9.1 Campo Incremental (AddIncrementField)

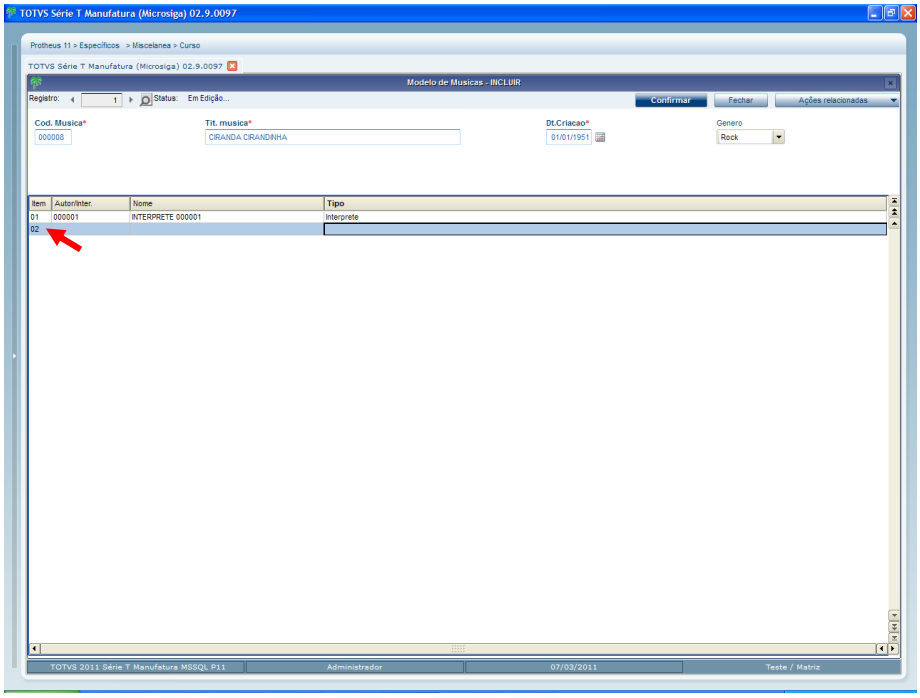
Podemos hacer que un campo de modelo de datos (Model) que parta de un componente de grid, pueda ser incrementado unitariamente a cada nueva línea insertada.
Por ejemplo, imaginemos el Pedido de Vendas, los ítems, ~~o~~ el número de ítem puede ser un campo incremental.

Para esto utilizamos el método ***AddIncrementField***. Ejemplo:

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

~~Donde~~ El ***VIEW_ZA2*** es un identificador (ID) de componente de la interfaz (*View*) donde se encuentra el campo, y el ***ZA2_ITEM*** es el nombre del campo que ~~será incrementado~~se incrementará.

Visualmente tenemos:



Importante: Este comportamiento ~~solo acontece~~solamente ocurre cuando la aplicación está siendo utilizada por la interfaz (*View*). Cuando el modelo de datos ~~es usado~~se utiliza directamente (*Web Services*, rutinas automáticas, etc.) el campo incremental tiene que ser informado normalmente.

9.2 Creación de botones en la barra de botones (AddUserButton)

Para la creación de botones adicionales en la barra de botones de la interfaz utilizamos el método ***AddUserButton***.

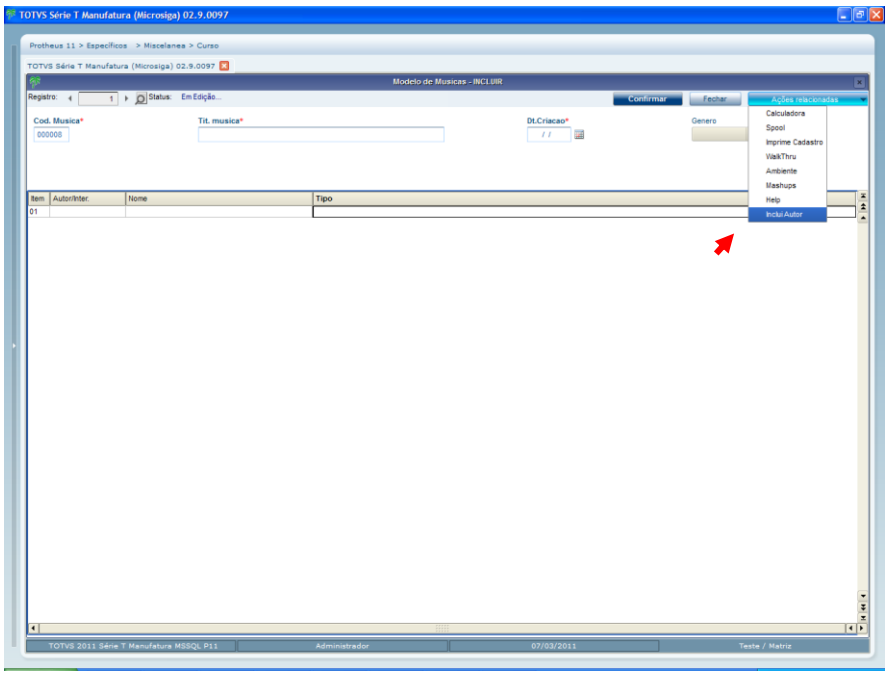
Ejemplo:

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

Donde ~~el~~ la opción **Inclui Autor**~~,~~ es el texto que ~~será presentado~~se mostrará en el botón, **CLIPS** es el nombre de la imagen del RPO[2] que ~~será usada para~~se utilizará en el botón, y ~~el~~ 3er parámetro ~~y~~ es el bloque de código que ~~será ejecutado~~se ejecutará al seleccionar el botón.

[2] RPO - Repositorio ~~do~~ del Microsiga Protheus para aplicaciones e imágenes.

Visualmente tenemos:



9.3 Título del componente (EnableTitleView)

En MVC podemos ~~asignar~~ atribuir un título para identificar cada componente de la interfaz, para esto utilizamos el método ***EnableTitleView***.

Ejemplo:

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

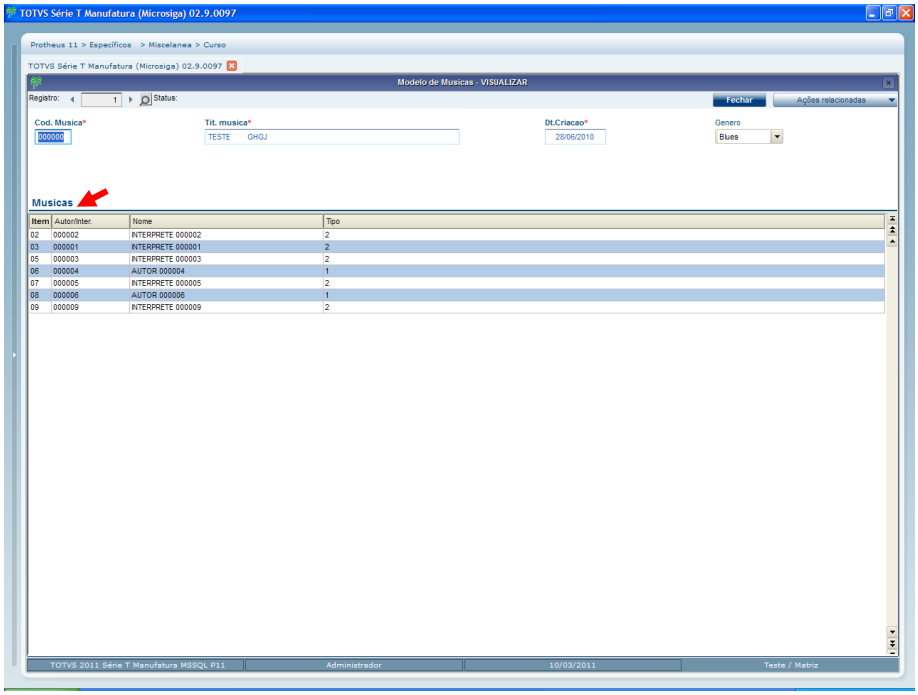
Donde ***VIEW_ZA2*** es el identificador (ID) del componente de la interfaz (*View*)~~,~~ y ***'MUSICAS'*** el titulo que desea para el componente.

También puede utilizar:

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

Donde el titulo que ~~será mostrado~~se mostrará es el que ~~fue definido~~se definió en el método ***SetDescription*** del modelo de datos (*Model*) para el componente.

Visualmente tenemos:



9.4 Edición de Campos en el componente de grid (SetViewProperty)

Una nueva característica que el *MVC* tiene, en el uso de la interfaz, es para que un componente de *grid*~~,~~ ~~hacer~~ haga ~~lo mismo durante la~~al mismo tiempo la edición de datos directamente en el *grid* y/ó en una pantalla del layout de formulario.

Para esto utilizamos el método ***SetViewProperty***. Este método habilita algunos comportamientos específicos en el componente de la ~~interface~~ interfaz (*View*), ~~conforme a~~de acuerdo con la directiva recibida.

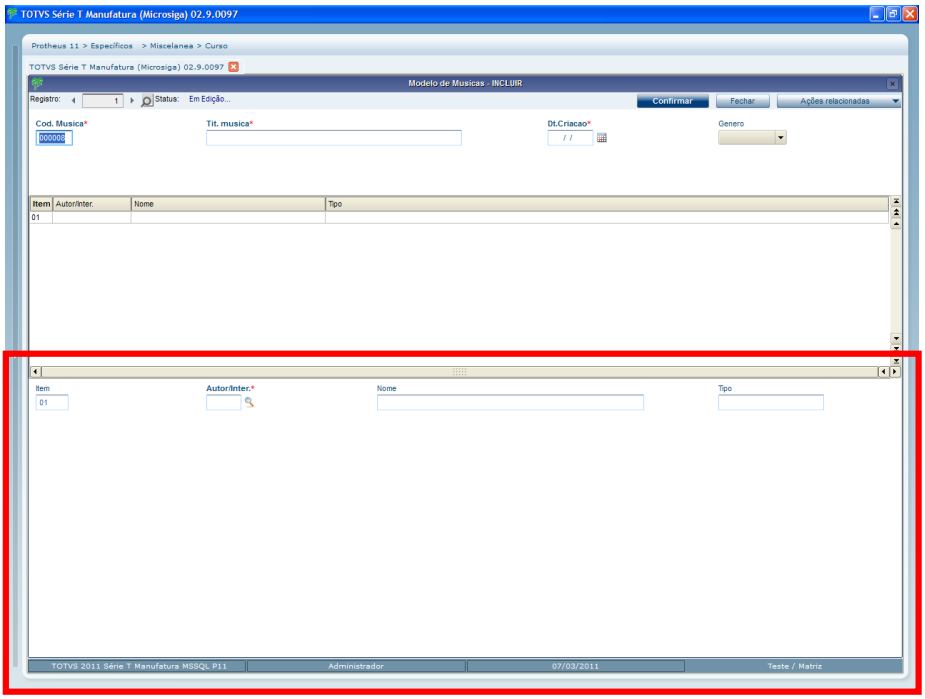
Ejemplo:

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

~~Donde~~ El ***VIEW_ZA2*** es el identificador (*ID*) del componente de la interfaz (View), donde se encuentra el campo, y ***ENABLEDGRIDDETAIL*** es la directiva que habilita el comportamiento.

{60} es el ~~porcentual~~ porcentaje que el formulario de edición ocupará del tamaño que el componente de *grid* ocupa actualmente. ~~Ejemplificando numéricamente~~Un ejemplo numérico, si para el componente del *grid* fue definido que se utilizará el 50% de la pantalla, al ~~-~~colocar 60 (60%) en el parámetro, se quiere indicar que ~~los~~ el 50% ~~son destinados~~se destina al componente de *grid*, el 60% ~~serán usados~~se utilizará para el formulario de edición.

Visualmente tenemos:



9.5 Creación de ~~folders~~ carpetas (CreateFolder)

En *MVC* podemos crear ~~folders~~ carpetas donde serán colocados los componentes de la ~~interface~~ interfaz (*View*).

Para esto utilizamos el método ***CreateFolder***.

Ejemplo:

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

Debemos dar un identificador (*ID*) para cada componente de la interfaz (View). ***PASTAS CARPETAS*** es un identificador (*ID*) dado que se da al folder a la carpeta.

Para la creación de la pestañaDespués de crear la carpeta principal, necesitamos es necesario crear las pestañas solapas de este folderesta carpeta. Para esto utilizamos el método ***AddSheet***.

Por ejemplo, crearemos 2 pestañassolapas:

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

Donde En que ***PASTAS CARPETAS*** es el identificador (*ID*) de folderla carpeta, y ***ABA01*** - ***ABA02*** son los IDs dados a cada pestaña solapa y ***Cabeçalho Encabezado*** e ***Ítem*** son los títulos de cada pestañasolapa.

Para que podamos colocar un componente en una pestañasolapa, necesitamos crear un box, un objeto, para recibir los elementos de la interfaz (*View*).

La forma para crear un ***box*** en una pestaña solapa es:

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

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

Debemos dar un identificador (ID) para cada componente de la interfaz (View).

- ***SUPERIOR*** e ***INFERIOR*** son los *ID*s dados que se da a cada ***box***.
- **100** indica el porcentual porcentaje que el ***box*** ocupará en la pestañasolapa.
- ***PASTAS*** es el identificador (*ID*) del folderde la carpeta.
- ***ABA01*** y ***ABA02*** son los *ID*s de las pestañassolapas.

Necesitamos relacionar el componente de la interfaz (*View*) con un **box** para mostrarlo, para esto usamos el método ***SetOwnerView***.

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

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

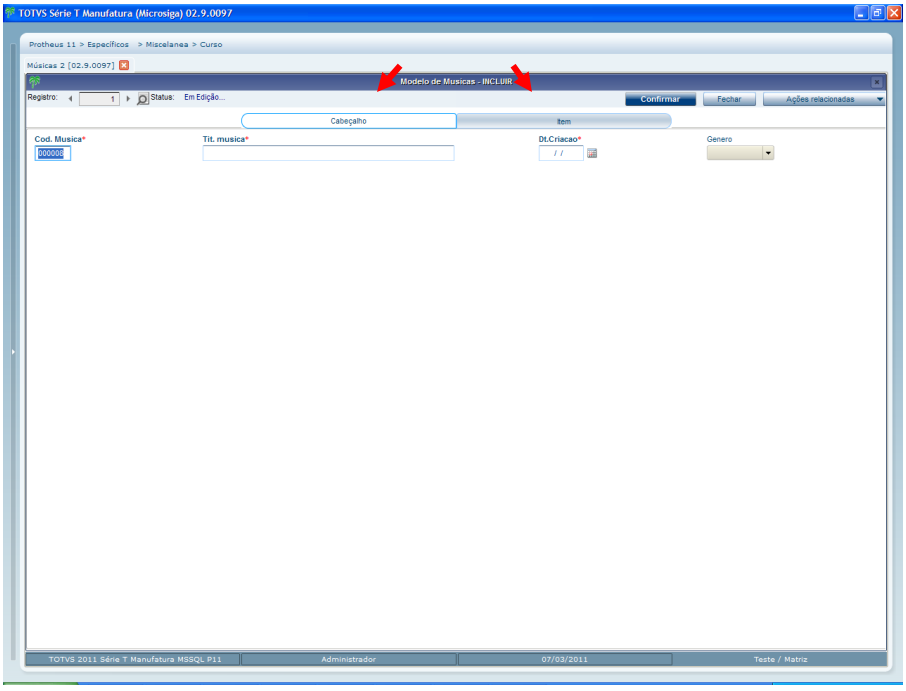
Resumiendo:

```
// Crear Folder en la view

oView:CreateFolder( 'PASTAS' )


// Crear pestañas en los folders

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

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

// Crear "box" horizontal para recibir algún elemento de la view
oView:CreateHorizontalBox( 'GERAL' , 100,,, 'SUPERIOR', 'ABA01' )
oView:CreateHorizontalBox( 'CORPO' , 100,,, 'INFERIOR', 'ABA02' )


// Relaciona el identificador (ID) de la View con un "box" para mostrarlo
oView:SetOwnerView( 'VIEW_ZA1' , 'SUPERIOR')
oView:SetOwnerView( 'VIEW_ZA2' , 'INFERIOR' )
```

Cuando ~~los folder~~las carpetas ~~son definidos~~se definen ~~–~~utilizando ~~–~~los ~~metadados~~ metadatos (diccionarios), automáticamente la interfaz (*View*) crea ~~estos folders~~estas carpetas. Si el componente colocado en una de las ~~pestañas~~ solapas tiene ~~folders~~ carpetas ~~definidos~~ definidas en los ~~metadados~~metadatos, ~~estos folders~~dichas carpetas ~~serán creados~~se crearán dentro de la ~~pestaña~~ solapa donde ~~él~~ éste se encuentra.

Visualmente tenemos:



9.6 ~~Agrupamiento~~ Agrupación de campos (AddGroup)

Una nueva característica que el *MVC* tiene para el uso de la interfaz, para un componente de formulario, es hacer ~~un agrupamiento~~una agrupación de los campos en la pantalla.

Por ejemplo, en un ~~cadastro~~ archivo de clientes podemos tener los campos para la dirección de entrega, correspondencia y facturación. Para ~~una visualización~~visualizar mejor podríamos agrupar los campos de cada dirección.

Para esto usamos el método ***AddGroup***.

Ejemplo:

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

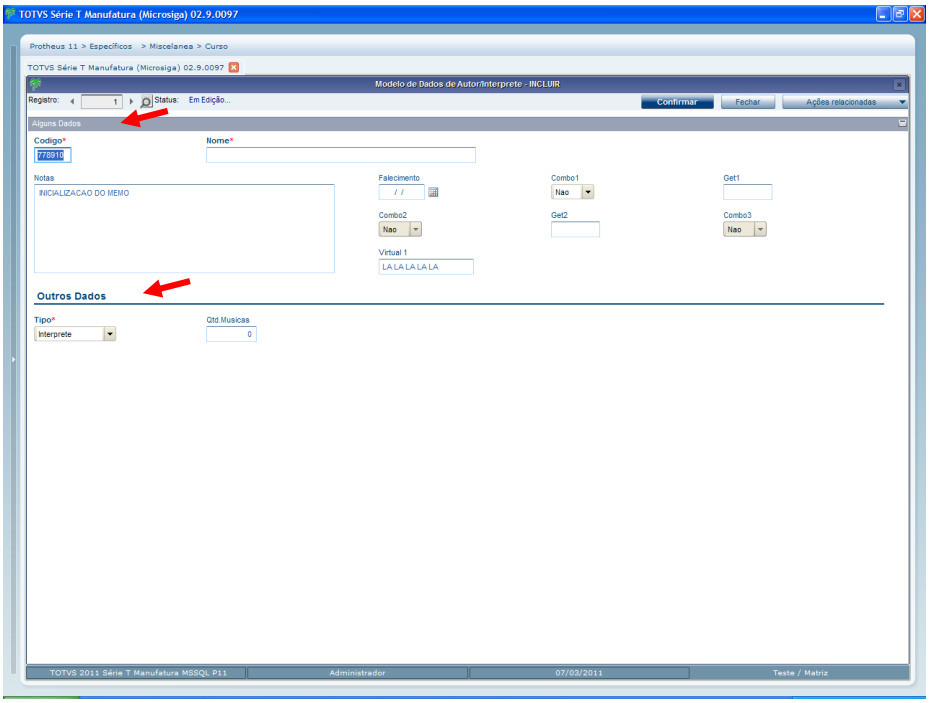
Debemos dar un identificador (ID) para cada componente de interfaz (View).

GRUPO01 es el identificador (ID) dado ~~al agrupamiento~~a la agrupación, el 2do parámetro es el título que ~~será presentado en el agrupamiento~~se presentará en la agrupación, el 1 es el tipo de ~~agrupamiento~~agrupación, puede ser 1- Ventana; 2 -Separador

Con ~~el agrupamiento creado~~la agrupación creada, ~~necesitamos decir~~es necesario indicar ~~cuales~~ los campos ~~serán~~ que formarán parte de ~~este agrupamiento~~esta agrupación. Para ~~esto~~ ello modificamos una propiedad de la estructura de algunos campos. Usaremos el método ***SetProperty***, que se verá más detalladamente en otro capítulo.

```
// Colocando todos los campos para un agrupamiento'
oStruZA0:SetProperty( '*', MVC_VIEW_GROUP_NUMBER, 'GRUPO01' )
// Cambiando el agrupamiento de algunos campos
oStruZA0:SetProperty( 'ZA0_QTDMUS',
MVC_VIEW_GROUP_NUMBER, 'GRUPO02' )
oStruZA0:SetProperty( 'ZA0_TIPO' ,
MVC_VIEW_GROUP_NUMBER, 'GRUPO02' )
```

Visualmente tenemos:



Observación: ~~Los agrupamientos~~Las agrupaciones ~~serán mostrados~~se mostrarán en la interfaz (View) en el orden de su creación.

9.7 Acción de la interfaz (SetViewAction)

Existe en MVC la posibilidad de ~~-~~ejecutar una función en algunas acciones de la interfaz (View). Este recurso puede ser utilizado cuando queremos ejecutar algo en la interfaz y que no tiene reflejo en el modelo de datos (Model) como un Refresh de pantalla por ejemplo.

Esto es posible en las siguientes acciones:

- *Refresh* de la interfaz~~;~~.
- Selección del botón ~~-~~confirmar de la interfaz~~;~~.
- Selección del botón ~~cancelar~~ anular de la interfaz~~;~~.
- Eliminación de la línea del *grid*~~;~~.
- Restauración de la línea del *grid*~~;~~.

Para esto usamos el método ***SetViewAction*** su sintaxis ~~-~~es:

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

Donde:

cActionlID ID del punto donde la acción será ejecutada, pueden ser:

REFRESH Ejecuta la acción del *Refresh* en la View~~;~~.

BUTTONOK Ejecuta la acción en la selección del botón confirmación de la View~~;~~.

BUTTONCANCEL Ejecuta la acción en la selección del botón ~~cancelar~~ anular de la View~~;~~.

DELETELINE Ejecuta ~~en~~ la acción ~~en la eliminación de~~al borrar la línea en el *grid*~~;~~.

UNDELETELINE Ejecuta la acción ~~en la restauración de~~al restaurar la línea en el *grid*~~;~~.

bAction Bloque con la acción a ~~ser ejecutada~~ejecutar. Recibe como parámetro:

REFRESH Recibe como parámetro el objeto View~~;~~.

BUTTONOK Recibe como parámetro el objeto de View~~;~~.

BUTTONCANCEL Recibe como parámetro el objeto de View~~;~~.

DELETELINE Recibe como parámetro el objeto de View, identificador (ID) de la View y número de la línea.

UNDELETELINE Recibe como parámetro el objeto de View, identificador (ID) de la View y número de la línea.

Ejemplo:

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

Importante: Estas acciones son ejecutadas cuando existe una interfaz (*View*). Lo que no ocurre cuando tenemos la instancia directa del modelo, rutina automática ~~ó~~ o *Web Service*. Se debe evitar entonces colocar en estas funciones acciones que

podrán influenciar la regla de negocio, ~~desde la ejecución de la aplicación sin interfaz estas acciones~~ya que al ejecutar la aplicación sin la interfaz dichas acciones no serán ejecutadas.

9.8 Acción ~~del campo en la interfaz~~de interfaz del campo (SetFieldAction)

Existe en *MVC* la posibilidad de ejecutar una función desde la validación del campo de algún componente del modelo de datos (*Model*).
Este recurso puede ser utilizado cuando queremos ejecutar algo en la interfaz y que no tiene reflejo en el modelo, como un ***Refresh*** de la pantalla ~~ó~~ abrir una pantalla auxiliar, por ejemplo.

Para esto utilizamos el método en ***SetFieldAction***.

Sintaxis:

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

Donde:

cIDField ID del campo (nombre):

bAction Bloque de la acción a ser ejecutada, recibe como parámetro:

- Objeto de la View
- El identificador (ID) de la View
- El identificador (ID) del Campo
- Contenido del Campo

Ejemplo:

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

Importante:

- Estas acciones son ejecutadas después de la validación del campo.
- Estas acciones son ejecutadas solo cuando existe una interfaz (*View*). Lo que no ocurre cuando tenemos la instancia directa del modelo, rutina automática ~~ó~~ *Web Service*.
- Se debe evitar entonces colocar en estas funciones acciones que puedan influenciar la regla de negocio, desde la ejecución de una aplicación sin interfaz esas acciones no serán ejecutadas.

9.9 Otros objetos (AddOtherObjects)

En la construcción de algunas aplicaciones puede ser que tengamos que agregar a la interfaz un componente que no sea parte de la interfaz estándar del MVC, como un gráfico, un calendario, etc.

Para esto utilizaremos el método ***AddOtherObject*** Sintaxis:

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

Donde el 1er parámetro es el identificador (ID) del ***AddOtherObjects*** y 2do parámetro es el bloque de código que será ejecutado para la creación de los otros objetos.
En *MVC* se limita a hacer la llamada de la función, la responsabilidad de construcción y actualización de los datos es del desarrollador de la función.

Ejemplo:

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

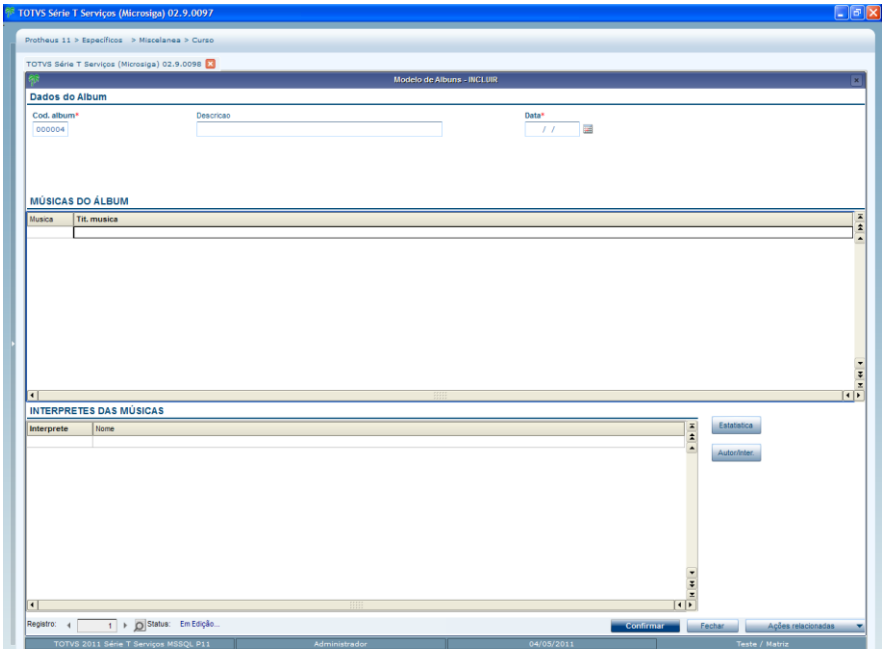
Note que el 2do parámetro recibe como parámetro un objeto que es el ***contenedor*** donde el desarrollador debe colocar sus objetos.

A continuación sigue un ejemplo del uso del método, donde colocamos en una sección de la interfaz (*View*) 2 botones. Observe los comentarios en el fuente:

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

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

// Crear "box" horizontal para recibir algún elemento de la view
oView:CreateHorizontalBox( 'EMCIMA' , 20 )
oView:CreateHorizontalBox( 'MEIO' , 40 )
oView:CreateHorizontalBox( 'EMBAIXO', 40 )

// Dividir en 2 "box" vertical para recibir un elemento de la view
oView:CreateVerticalBox( 'EMBAIXOESQ', 20, 'EMBAIXO' )
oView:CreateVerticalBox( 'EMBAIXODIR', 80, 'EMBAIXO' )

// Relacionar el identificador (ID) de la View con un "box" para mostrarlo
oView:SetOwnerView( 'VIEW_ZA3', 'EMCIMA'                 )
oView:SetOwnerView( 'VIEW_ZA4', 'MEIO'                   )
oView:SetOwnerView( 'VIEW_ZA5', 'EMBAIXOESQ')
// Liga a identificación del componente
oView:EnableTitleView( 'VIEW_ZA3' )
```

```
oView:EnableTitleView( 'VIEW_ZA4', "MÚSICAS DO ÁLBUM"                )
oView:EnableTitleView( 'VIEW_ZA5', "INTERPRETES DAS MÚSICAS" )

// Agrega un objeto externo a la View del MVC
// AddOtherObject(cFormModelID,bBloco)
// cIDObject - Id

// bBloco - Bloco llamado, deberá ser utilizado para  crear los objetos de la
pantalla  externa MVC.
oView:AddOtherObject("OTHER_PANEL", {|oPanel| COMP23BUT(oPanel)})

// Asocia el box que mostrará los otros objetos
oView:SetOwnerView("OTHER_PANEL",'EMBAIXODIR')

Return oView

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

// Anclamos los objetos en el oPanel pasado
@ 10, 10 Button 'Estatistica'     Size 36, 13 Message 'Contagem da FormGrid' Pixel
Action COMP23ACAO( 'ZA4DETAIL', 'Existem na Grid de Musicas' ) of oPanel

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

Return NIL
```

Visualmente tenemos:



10. Tratamientos de estructuras de datos

Como se dijo anteriormente, en *MVC* no se trabaja vinculado a los metadatos de Microsiga Protheus, ~~el~~ este trabaja vinculado a estructuras. Estas estructuras, a su vez pueden ser construidas a partir de ~~metadados~~ metadatos (diccionarios).

Veremos algunos tratamientos que pueden ser realizados en las estructuras conforme a las necesidades.

10.1 Selección de campos para la estructura (FWFormStruct)

Ahora crearemos una estructura basada en ~~metadados~~ metadatos (diccionarios), utilizando la función ***FWFormStruct***, esta ~~lleva~~ toma en consideración todos los campos de ~~las entidades~~de los entes, respetando nivel, módulo, uso , etc.

Si quisiéramos seleccionar cuales son los campos de los ~~metadados~~ metadatos (diccionarios) que serán parte de la estructura, debemos utilizar el 3er parámetro del ***FWFormStruct***, que es un bloque de código que será ejecutado para cada campo, la función traerá los metadatos (diccionarios) y recibe como parámetro el nombre del campo.

El bloque de código debe retornar un valor lógico, ~~donde~~ si es .T. (verdadero) el campo será parte de la estructura, si es .F. (falso) el campo no será parte de la estructura.

Ejemplo:

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

Donde la función puede ser:

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

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

```
Return lRet
```

En el ejemplo la función anterior el campo ***ZA0_QTDMUS*** no será parte de la estructura.

El diccionario de campos (SX3) ~~de~~ del ~~metadatos es posicionado~~metadato se posiciona para cada campo.

Importante: Este tratamiento puede ser realizado tanto para las estructuras que serán utilizadas en el modelo de datos (*Model*) como ~~tanto para en~~para la interfaz (*View*).

Tenga el siguiente cuidado: Si un campo obligatorio fue eliminado de la estructura de la interfaz (*View*), y no está siendo mostrado para el usuario, el modelo de datos (*Model*) hará la validación diciendo que un campo obligatorio no fue informado.

10.2 ~~Eliminar~~ Eliminación de campos de una estructura (RemoveField)

Una forma para eliminar un campo de una estructura es utilizar el método ***RemoveField***.

Ejemplo:

```
Local oStruZA0 := FWFormStruct( 2, 'ZA0')
oStruZA0: RemoveField('ZA0_QTDMUS')
```

En el ejemplo anterior el campo ***ZA0_QTDMUS*** fue eliminado de la estructura.

Importante: Este tratamiento puede ser realizado tanto para las estructuras que serán utilizadas en el modelo de datos (*Model*) tanto como en la interfaz (*View*).

Tome el siguiente cuidado: ~~Si fue eliminado de la estructura de la interfaz (View) un campo obligatorio, y no está siendo mostrado para el usuario~~Si un campo obligatorio se elimina de la estructura de la interfaz (View) y no se muestra al usuario, el modelo de dados (*Model*) realizará la validación ~~diciendo~~ indicando que ~~un campo obligatorio no fue informado~~no se informó un campo obligatorio.

10.3 ~~Modificar~~ Modificación de las propiedades de un campo (SetProperty)

Ahora construiremos una estructura basada en los metadatos (diccionarios), utilizando la función ***FWFormStruct***, son respetadas las propiedades que el campo tiene como validación, inicializador estándar y el modo de edición, etc.

Si hubiera necesidad de modificar alguna propiedad del campo en la estructura, utilizaremos el método ***SetProperty***.

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

Donde, el 1er parámetro es el nombre del campo que se desea modificar ~~ó~~ o ~~asignar~~ atribuir una propiedad el 2do es ~~-~~la propiedad que es modificada o asignada y el 3ero es el valor para la propiedad.

En el ejemplo anterior el campo ***ZA0_QTDMUS*** solo podrá ser editado en la operación de inclusión.

Las propiedades para los campos de la estructura del modelo de datos (*Model*) son:

Propiedades para los campos de estructura del modelo de datos (*Model*)	**Tipo**	**Descripción**
MODEL_FIELD_TITULO	C	Titulo
MODEL_FIELD_TOOLTIP	C	Descripción completa del campo
MODEL_FIELD_IDFIELD	C	Nombre (ID)
MODEL_FIELD_TIPO	C	Tipo
MODEL_FIELD_TAMANHO	N	Tamaño
MODEL_FIELD_DECIMAL	N	Decimales
MODEL_FIELD_VALID	b	Validación
MODEL_FIELD_WHEN	B	Modo de edición
MODEL_FIELD_VALUES	A	Lista de valores permitido del campo (combo)
MODEL_FIELD_OBRIGAT	L	Indica si el campo es obligatorio
MODEL_FIELD_INIT	B	Inicializador estándar
MODEL_FIELD_KEY	L	Indica si el campo es llave
MODEL_FIELD_NOUPD	L	Indica si el campo puede recibir valor en una operación de *Update*.
MODEL_FIELD_VIRTUAL	L	Indica si el campo es virtual

Las propiedades para los campos de la estructura de la interfaz (View) son:

Propiedades para campos de estructura de la interfaz (View)	**Tipo**	**Descripción**

MVC_VIEW_IDFIELD	C	Nombre del Campo
MVC_VIEW_ORDEM	C	Orden
MVC_VIEW_TITULO	C	Titulo del campo
MVC_VIEW_DESCR	C	Descripción del campo
MVC_VIEW_HELP	A	Array con el Help
MVC_VIEW_PICT	C	Picture
MVC_VIEW_PVAR	B	Bloque de Picture Var
MVC_VIEW_LOOKUP	C	Consulta F3
MVC_VIEW_CANCHANGE	L	Indica si el campo es editable
MVC_VIEW_FOLDER_NUMBER	C	Folder del campo
MVC_VIEW_GROUP_NUMBER	C	~~Agrupamiento~~ Agrupación del campo
MVC_VIEW_COMBOBOX	A	Lista de valores permitido del campo (Combo)
MVC_VIEW_MAXTAMCMB	N	Tamaño Máximo de la mayor opción del combo
MVC_VIEW_INIBROW	C	Inicializador de Browse
MVC_VIEW_VIRTUAL	L	Indica ~~se~~ si el campo es virtual
MVC_VIEW_PICTVAR	C	Picture Variable

Los nombres de propiedades citados en las tablas son directivas de compilación de tipo ***#DEFINE.***

Para utilizar este ***#DEFINE*** es necesario incluir la siguiente directiva en el fuente:

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

También es posible ~~asignar~~ atribuir una propiedad ~~para~~ a todos los campos de una estructura utilizando en el nombre del campo un asterisco "*"

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

10.4 Creación de campos adicionales en una estructura (AddField)

~~Si se quisiera creas~~Si queremos crear un campo en una estructura ~~ya existente~~que ya existe, utilizamos el método ***Addfield***.

Hay diferencias en la secuencia de parámetros de este método ~~para agregar~~cuando se agregan campos ~~para~~ a la estructura del modelo de datos (*Model*) ~~ó~~ o de la interfaz (*View*).

La sintaxis para el modelo de datos (Model) es:

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

Donde:

cTitulo	Título del campo~~;~~
cTooltip	Tooltip del campo~~;~~
cIdField	Id del Field~~;~~
cTipo	Tipo del campo~~;~~
nTamanho	Tamaño del campo~~;~~
nDecimal	Decimal del campo~~;~~
bValid	Code-block de validación del campo~~;~~
bWhen	Code-block de validación del modo de edición del campo~~;~~
aValues	Lista de valores permitido del campo~~;~~
lObrigat	Indica si el campo es obligatorio~~;~~
bInit	Code-block de inicialización del campo~~;~~
lKey	Indica si se trata de un campo llave~~;~~
lNoUpd	Indica si el campo no puede recibir valor en una operación de Update~~;~~
lVirtual	Indica si el campo es virtual~~;~~

A continuación mostramos un ejemplo del uso:

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

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

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

```
NIL , ;                                          // [08] B Code-block de validação
When do campo
{'1=Sim','2=Não'} , ;                            // [09] A Lista de valores permitido
do campo
NIL , ;                                          // [10] L Indica se o campo tem
preenchimento obrigatório
FwBuildFeature( STRUCT_FEATURE_INIPAD, "'2'" ) , ; // [11] B Code-block de
inicializacao do campo

NIL , ;                                          // [12] L Indica se trata de um
campo chave
NIL , ;                                         // [13] L Indica se o campo pode
receber valor em uma operação de update.
             .T. )

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

La sintaxis para la interfaz (View) es:

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

Donde:

cIdField	Nombre del Campo;
cOrdem	Orden;
cTitulo	Título del campo;
cDescric	Descrición completa del campo;
aHelp	Array con Help;
cType	Tipo de campo;
cPicture	Picture;
bPictVar	Bloco de PictureVar;
cLookUp	Consulta F3;
lCanChange	Indica si el campo es editable;
cFolder	Carpeta del campo;
cGroup	Agrupamiento del campo;
aComboValues	Lista de valores permitido del campo (combo);
nMaxLenCombo	Tamaño máximo de la mayor opción del combo;
cIniBrow	Inicializador de Browse;

lVirtual Indica si el campo es virtual~~;~~

cPictVar Picture Varible~~;~~

Ejemplo de su uso:

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


oStruZA0:AddField( ;                    // Ord. Tipo Desc.
'ZA0_XEXEM1' , ;                        // [01]   C     Nombre del Campo
'50', ;                                 // [02]   C     Orden
AllTrim( 'Exemplo 1'),;                 // [03]   C     Titulo del campo
AllTrim( 'Campo Exemplo 1' ),;          // [04]   C     Descripción do campo
{ 'Exemplo de Campo de Manual 1' } , ;// [05]   A     Array com Help
'C', ;                                  // [06]   C     Tipo del campo
'@!' , ;                                // [07]   C     Picture
NIL, ;                                  // [08]   B     Bloco de Picture Var
'', ;                                   // [09]   C     Consulta F3
.T., ;                                  // [10]   L     Indica si el  campo es editable
NIL, ;                                  // [11]   C     Folder del campo
NIL, ;                                  // [12]   C     Agrupamiento del campo
{'1=Sim','2=Não'}, ;                    //  [13]    A       Lista de valores permitido del
campo(Combo)
NIL, ;                                  // [14]    N       Tamaño Máximo de la mayor opción
del combo
NIL, ;                                  // [15]   C     Inicializador de Browse
.T., ;                                  // [16]   L     Indica si el campo es virtual
NIL)                                    // [17]   C     Picture Variable
```

Obs.: Los campos de tipo lógico serán mostrados como un checkbox en la interfaz (*View*)

10.5 Formato del bloque de código para una estructura (FWBuildFeature)

Algunas propiedades de los campos de la estructura necesitan una construcción específica del bloque de código. Si ~~se asigna o manipulan esas propiedades estas deben ser informadas~~dichas propiedades se atribuyen o manejan, estas se deben informar ~~como~~ en el estándar que el *MVC* espera.

Si se tratan esas propiedades para el uso en aplicaciones se debe utilizar la función ***FWBuildFeature*** para construirla.

Ejemplo:

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

Donde el 1er parámetro indica cual es la propiedad a ser construida, el 2do es el contenido ~~a ser asignado~~que se atribuirá. El 2do parámetro siempre debe retornar un dato ~~de~~ del tipo carácter.

Las propiedades que necesitan ser tratadas con esta función son:

STRUCT_FEATURE_VALID Para la **validación**

Formatado: Português (Brasil)

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

STRUCT_FEATURE_WHEN Para el **modo de edición**

STRUCT_FEATURE_INIPAD Para el **inicializador estándar**

STRUCT_FEATURE_PICTVAR Para **PictureVar**

Los nombres de propiedades citados son ***#DEFINE***. Para utilizar este ***#DEFINE*** es necesario incluir la siguiente directiva en el fuente:

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

Obs.: Utilice siempre la ~~-~~función ***FWBuildFeature*** para ~~la construcción de~~crear las propiedades~~,~~ de lo contrario ~~podrán~~ pueden ocurrir errores en la aplicación, tal como en la actualización de las variables de memoria para los componentes de formulario.

10.6 Campos de tipo MEMO virtuales (FWMemoVirtual)

Algunos campos de tipo MEMO utilizan tablas para la grabación de sus valores (SYP[3]), esos campos deben ser informados en la estructura para que en MVC se pueda hacer su tratamiento correctamente.

Para esto utilizamos la función ***FWMemoVirtual***.

Ejemplo:

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

[3] SYP - Tabla de Microsiga Protheus que almacena los datos de los campos de tipo MEMO virtuales

Para estos campos MEMO siempre debe ~~hacer~~ haber otro campo que contenga el código con que el campo MEMO fue almacenado en la tabla auxiliar.

En el ejemplo, ***oStruZA1*** es una estructura que contiene los campos MEMO y el segundo parámetro un vector bi-dimensional donde cada par relaciona el campo de la estructura que contiene el código del campo MEMO con el campo MEMO.

Si la tabla auxiliar no va a ser utilizada por la SYP, un 3er parámetro deberá ser pasado en un vector bi-dimensional, como el alias de la tabla auxiliar.

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

Observación: Tanto el campo MEMO como el campo que almacenará su código deben ser parte de la estructura.

10.7 Creación manual del gatillo (AddTrigger / FwStruTrigger)

Si quisiéramos agregar un gatillo a una estructura ya existente, utilizamos el método ***AddTrigger***

La sintaxis es:

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

Donde:

cIdField	Nombre (ID) del campo de origen~~;~~
cTargetIdField	Nombre (ID) del campo de destino~~;~~
bPre	Bloque de código de validación de la ejecución del gatillo~~;~~
bSetValue	Bloque de código de ejecución del gatillo~~;~~

Los bloques de código de este método tienen una construcción específica. Si se quieren ~~asignar o manipular~~atribuir o manejar estas propiedades, deben ser informadas en el estándar que *MVC* espera.

Para facilitar la construcción de un gatillo fue creada la función ***FwStruTrigger***, esta retorna un array con 4 elementos ya con el formato para el uso en ***AddTrigger***.

La sintaxis es:

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

Donde:

cDom	Campo Dominio~~;~~
cCDom	Campo de Contra dominio~~;~~
cRegra	Regla de llenado~~;~~
lSeek	Se posicionara o no antes de la ejecución de los ~~gatilhos~~gatillos~~;~~
cAlias	Alias de la tabla a ser posicionada~~;~~
nOrdem	Orden de la tabla a ser posicionada~~;~~
cChave	~~Llave~~ Clave de búsqueda de la tabla a ser posicionada~~;~~
cCondic	Condición para ejecutar el gatillo~~;~~

Ejemplo:

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

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

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

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

10.8 Cómo Rretirar ~~los folders~~las carpetas de una estructura (SetNoFolder)

Se quisiéramos retirar las carpetas que están configuradas en una estructura, por ejemplo, por el uso de la función ***FWFormStruct***, usamos el método ***SetNoFolder***. De la siguiente forma:

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

// Retira los folders da estructura
oStruZA0:SetNoFolder()
```

10. 9 Cómo ~~Retirar~~ retirar ~~los agrupamientos~~las agrupaciones de campos de una estructura (SetNoGroups)

Si quisiéramos retirar ~~los agrupamientos~~las agrupaciones de campos que están configuradas en una estructura, por ejemplo, cuando usamos la función ***FWFormStruct***, usamos el método ***SetNoGroups***. De la siguiente forma:

```
Local oStruZA0   := FWFormStruct( 2, 'ZA0' )
// Retira los agrupamientos de campos de la estructura
```


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

```
oStruZA0:SetNoGroups()
```

11. Creación de campos de total ~~ó~~ o contadores (AddCalc)

En *MVC* es posible crear automáticamente un nuevo componente, compuesto de campos totalizadores ~~ó~~ o contadores, estos son componentes de cálculos.

Los campos de componente de cálculos son basados en componentes de *grid* del modelo. Actualizando el componente del *grid* automáticamente los campos del componente de cálculos serán actualizados.

El ***Addcalc*** es un componente de modelo de datos (*Model*) responsable por esto.

La sintaxis es:

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

Donde:

cId Identificador del componente de cálculos~~.~~

cOwner Identificador del componente superior (*owner*).
No necesariamente es el componente de *grid* de donde varían los datos.
Normalmente el superior es *AddField* principal del modelo de datos (*Model*)~~.~~

cIdForm Código del componente de grid que contiene el campo, a que se refiere el campo calculado~~.~~

cIdField Nombre del campo del componente de *grid* a que se refiere el campo calculado~~.~~

cIdCalc Identificador (nombre) para el campo calculado~~.~~

cOperation Identificador de la operación a ser realizada.

Las operaciones pueden ser:

SUM	Hace la suma de campo del componente de *grid*~~.~~
COUNT	Hace el ~~conteo~~ cómputo de campo del componente de *grid*~~.~~
AVG	Hace el promedio del campo del componente de *grid*~~.~~
FORMULA	Ejecuta una fórmula para el campo del componente de *grid*~~.~~

bCond Condición para la validación del campo calculado. Recibe como parámetro el objeto del modelo. Retornando .T. (verdadero) ejecuta la operación del campo calculado~~.~~

Ejemplo: {|oModel| teste (oModel)}~~;~~

bInitValue Bloque de código para el valor inicial para el campo calculado. Recibe como parámetro el objeto del modelo~~;~~.

Ejemplo: {|oModel| teste (oModel)}~~;~~

cTitle Título para el campo calculado~~;~~.

bFormula Fórmula a ser utilizada cuando el parámetro cOperation es de tipo FORMULA.

Recibe como parámetros: el objeto del modelo, el valor actual del campo fórmula, el contenido del campo del componente del *grid*, campo lógico indicando si es una ejecución de suma (.T. (verdadero)) o resta (.F. (falso~~));~~)).

El valor retornado ~~será asignado~~se atribuirá al campo calculado~~;~~.

Ejemplo:

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

nTamanho ——Tamaño del campo calculado (Si no fue informado usa el tamaño estándar). Los —tamaños estándares para las operaciones son:

SUM —Será del tamaño del campo del componente de *grid* + 3;

Si el campo del componente de *grid* tiene ~~el~~ tamaño ~~de~~ 9, el campo calculado será 12.

COUNT Será de tamaño ~~-~~fijo en 6~~;~~.

AVG Será de tamaño del campo del componente del *grid*. Si el campo del componente del *grid* tiene ~~el~~ tamaño ~~de~~ 9, el campo calculado será 9~~;~~.

FORMULA Será el tamaño del campo del componente de grid + 3. Si el campo del componente de grid tiene ~~el~~ tamaño ~~de~~ 9, el campo calculado será 12~~;~~.

nDecimal Número de decimales del campo calculado~~;~~.

Observación: Para las operaciones de **SUM** y **AVG** el campo de componente del *grid* tiene que ser del tipo numérico.

Ejemplo:

```
Static Function ModelDef()

oModel:AddCalc( 'COMP022CALC1', 'ZA1MASTER', 'ZA2DETAIL', 'ZA2_AUTOR', 'ZA2__TOT01', 'COUNT', { | oFW | COMP022CAL( oFW, .T. ) },,'Total Pares' )
```

```
oModel:AddCalc( 'COMP022CALC1', 'ZA1MASTER', 'ZA2DETAIL',
'ZA2_AUTOR', 'ZA2__TOT02', 'COUNT', { | oFW | COMP022CAL(
oFW, .F. ) },,'Total Impares' )
```

Donde:

COMP022CALC1 Es el identificador del componente de cálculos~~;~~

ZA1MASTER Es el identificador del componente superior (*owner*)~~;~~

ZA2DETAIL Es el código del componente de *grid* de donde vienen los datos~~;~~

ZA2_AUTOR Es el nombre del campo de componente del *grid* a que se refiere el campo calculado~~;~~

ZA2__TOT01 Es el Identificador (nombre) para el campo calculado~~;~~

COUNT Es el Identificador de la operación a ser realizada~~;~~

{ | oFW | COMP022CAL(oFW, .T.) } Es la condición para la validación del campo calculado~~;~~

'Total Pares' Es el título para el campo calculado~~;~~.

En la **ViewDef** también tenemos que ~~hacer la definición del~~definir el componente de cálculo. Los datos utilizados en un componente de cálculo son basados en un componente de *grid*, pero, se muestran de la misma forma que un componente de formulario, para el cual ~~utilizarnos~~ utilizamos el componente de cálculo, **AddField**, y para obtener la estructura que fue creada en el **ModelDef** utilizamos **FWCalcStruct**.

Ejemplo:

```
Static Funcion View

// Crea objeto de la estructura
oCalc1 := FWCalcStruct( oModel:GetModel( 'COMP022CALC1') )
//Agrega en nuestra View un control de
//tipo FormGrid(antiguo newgetdados)
oView:AddField( 'VIEW_CALC', oCalc1, 'COMP022CALC1' )
```

12. Otras funciones para MVC

Algunas funciones pueden ser especialmente útiles durante el desarrollo de una aplicación.

12.1 Ejecución directa de la interfaz (FWExecView)

Hacer la ejecución de la interfaz (View) con una determinada operación.

Esta función ~~instancia~~ crea la ~~interface~~ interfaz (View) y por consiguiente el modelo de datos (*Model*) con las operaciones de **visualizar**, **incluir**, ~~**alterar**~~ **modificar** o ~~**excluir**~~**borrar**. La intención es hacer ~~similarmente~~ de manera similar lo que hacían las funciones **AXVISUAL**, **AXINCLI**, **AXALTERA** y **AXDELETA**.

Sintaxis:

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

Donde:

cTitulo Título de la ventana~~;~~

cPrograma Nombre del programa fuente~~;~~

nOperation Indica el código de operación (incluir, ~~alterar ó excluir~~modificar o borrar)~~;~~

oDlg Objeto de la ventana en que la View debe ser colocada. Si no es informado, una nueva ventana se ~~crearacrear~~á~~;~~

bCloseOnOK Indica si la ventana debe ser cerrada al final de la operación. Si retorna .T. (verdadero) cierra la ventana~~;~~

bOk Bloque ejecutado en la selección del botón confirmar, si retorna .F. (falso) impedirá que se cierre la ventana~~;~~

nPercReducao Si ~~es informado~~se informa reduce la ventana ~~porcentualmente~~en porcentaje~~;~~

aEnableButtons Indica los botones de la barra de botones que estarán habilitados~~;~~

bCancel Bloque ejecutado en la selección del botón ~~cancelar~~anular~~,~~

El retorno de esta función será:

0 Si el usuario finaliza la operación con el botón confirmar~~;~~

1 Si el usuario finaliza la operación con o botón ~~cancelar~~anular~~;~~

Ejemplo:

```
lOk             := ( FWExecView('Inclusão por FWExecView','COMP011_MVC',
MODEL_OPERATION_INSERT,,|| .T. ) == 0 )


If lOk
      Help( ,, 'Help',, 'Foi confirmada a operação, 1, 0 )
```

Formatado: Espanhol (México)

Formatado: Espanhol (México)

Formatado: Espanhol (México)

Formatado: Espanhol (México)

Formatado: Espanhol (México)

```
Else
        Help( ,, 'Help',, 'Foi cancelada a operação, 1, 0 )
EndIf
```

12.2 Modelo de dados activo (FWModelActive)

En una aplicación podemos trabajar con más de un modelo de datos (*Model*). La función **FWModelActive** retorna el modelo de datos (*Model*) que está activo en el momento.

Ejemplo:

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

Para definir cuál es el modelo de datos (*Model*) activo:

```
Local oModelBkp   := FWModelActive()

FWModelActive( oModelBkp )
```

12.3 ~~Interface~~ Interfaz activa (FWViewActive)

En una aplicación podemos trabajar con más de una interfaz (View). La función ***FWViewActive*** retorna la interfaz (View) que está activa en el momento.

Ejemplo:

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

Para definir cuál es la interfaz (*View*):

```
Local oViewBkp   := FWViewActive()

FWViewActive(oViewBkp)
```

12.4 Cargar el modelo de datos de una aplicación ya existente (FWLoadModel)

Para crear un objeto con el modelo de datos de una aplicación, utilizamos la función **FWLoadModel**.

Sintaxis:

Formatado: Português (Brasil)
Formatado: Português (Brasil)
Formatado: Português (Brasil)
Formatado: Espanhol (Espanha - tradicional)
Formatado: Espanhol (Espanha - tradicional)
Formatado: Espanhol (Espanha - tradicional)
Formatado: Espanhol (Espanha - tradicional)
Formatado: Espanhol (Espanha-internacional)

FWLoadModel(<nombre del fuente>)

Ejemplo:

```
Static Function ModelDef()
// Utilizando un model que ya existe en otra aplicación Return
FWLoadModel( 'COMP011_MVC' )
```

12.5 Cargar la interfaz de una aplicación ya existente (FWLoadView)

Para crear un objeto con el modelo de datos de una aplicación ya existente, utilizamos la función **FWLoadView**.

Sintaxis:

FWLoadView (<nombre del fuente>)

Ejemplo:

```
Static Function ViewDef()
// Utilizando una view que ya existe en otra aplicación
Return FWLoadView( 'COMP011_MVC' )
```

12.6 Cargar el menú de una aplicación ya existente (FWLoadMenudef)

Para crear un vector con las opciones de menú de una aplicación , utilizamos la función ***FWLoadMenudef***.

Sintaxis:

```
FWLoadMenudef ( <nombre del fuente> )
```

Ejemplo:

```
Static Function MenuDef()
// Utilizando un menú que ya existe en otra aplicación
Return FWLoadMenuDef( 'COMP011_MVC' )
```

12.7 Creación de un menú estándar (FWMVCMenu)

Podemos crear un menú con opciones estándar para MVC utilizando la función ***FWMVCMENU***.

Sintaxis:

```
FWMVCMENU ( <nombre del fuente> )
```

Ejemplo:

Formatado: Espanhol (Espanha-internacional)

Formatado: Espanhol (Espanha-internacional)

Formatado: Espanhol (Espanha - tradicional)

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

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

Será creado un menú estándar con las opciones: **Visualizar, Incluir, ~~Alterar~~Modificar, ~~Excluir~~Borrar, Imprimir y Copiar.**

13. Browse ~~columna con marcado~~con columna de marcado (FWMarkBrowse)

Si se desea construir una aplicación con ~~una~~ un *Browse* utilizando una columna para marcar, ***MarkBrowse*** funciona de manera similar a los tradicionales ADVPL, se utilizara la clase ***FWMarkBrowse***.

Asimismo como un ***FWmBrowse*** (ver cap. 0 3. Aplicaciones *Browses*), ***FWMarkBrowse*** no es exclusivamente del MVC también puede ser utilizado por las aplicaciones que no lo utilicen.

Este contenido no pretende profundizar sobre los recursos de ***FWMarkBrowse***, se darán a conocer sus principales funciones y características para su uso en aplicaciones *MVC*.

Como premisa, es necesario contar con un campo en la tabla de tipo carácter de tamaño 2 que recibirá físicamente la marca. Se genera una marca diferente cada vez que una ***FWMarkBrowse*** sea ejecutada.

Iniciar la construcción básica de un ***FWMarkBrowse***.

Primeramente se debe crear un objeto ***Browse*** de la siguiente manera:

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

Se define la tabla que aparece en ***Browse*** con el método ***SetAlias***. Las columnas, órdenes, etc., para mostrarlos, serán obtenidas a través de los metadatos (diccionarios)).

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

Se define un título que aparecerá como el método ***SetDescription***.

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

Se define cual será el campo de la tabla que recibirá la marca física.

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

Formatado: Português (Brasil)
Formatado: Português (Brasil)
Formatado: Português (Brasil)

Al final se activa la clase

```
oMark:Activate()
```

Con esta estructura básica se construye una aplicación con ***Browse***.

Por el momento solo se tiene una columna de marcado, ahora se tiene que definir una acción para los ítems marcados. Para esto, podemos colocar un ***MenuDef*** de la aplicación a la función que tratará a los ítems marcados.

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

En la función que tratará a los ítems marcados será necesario identificar si un registro esta ~~ó~~ o no marcado. Para saber la marca que está siendo utilizado en el momento, utilizamos el método ***Mark***.

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

Y para saber si el registro está marcado usamos el método ***IsMark*** pasando como parámetro la marca.

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

Es posible también colocar otras opciones, como visualizar ~~ó alterar~~o modificar en el menú de opciones (***MenuDef***), pero será necesario crear también el modelo de datos (*Model*) de la interfaz (*View*).

~~Todas las otras~~Las demás características del ***FWMBrowse*** también se aplican al ***FWMarkBrowse*** como leyendas, filtros, detalles, etc.

Un recurso que el ***FWMarkBrowse*** tiene, es el control ~~de marcación exclusiva~~del marcado exclusivo del registro por parte ~~el~~ del usuario.

Donde, si 2 usuarios abren el mismo ***FWMarkBrowse*** e ~~intentarán~~ intenta marcar el mismo registro, el ~~proprio~~ mismo ***FWMarkBrowse*** permitirá que ~~solo~~ sólo se ejecute ~~una marcación~~un marcado. Para habilitar esta característica usamos el método ***SetSemaphore***.

~~Abajo, sigue~~Ver a continuación un ejemplo de ***FWMarkBrowse***

```
User Function COMP025_MVC()
Private oMark


// Instanciamiento de la clase
oMark := FWMarkBrowse():New()


// Definición de la tabla a ser utilizada
oMark:SetAlias('ZA0')
```

```
// Define si utiliza control de marcación exclusiva del oMark:SetSemaphore(.T.)

// Define el titulo del browse de marcación
oMark:SetDescription('Selección del Cadastro de Autor/Interprete')
// Define el campo que será utilizado para a marcación
oMark:SetFieldMark( 'ZA0_OK' )

// Define a leyenda

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

// Definición del filtro de aplicación
oMark:SetFilterDefault( "ZA0_TIPO=='1'" )

// Activación de la clase
oMark:Activate()

Return NIL

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

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

Return aRotina

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

//---------------------- ---------------------------------------------
Static Function ViewDef()
// Utilizando uma View que ja existe
```

Formatado: Inglês (Estados Unidos)

Formatado: Inglês (Estados Unidos)

Formatado: Inglês (Estados Unidos)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

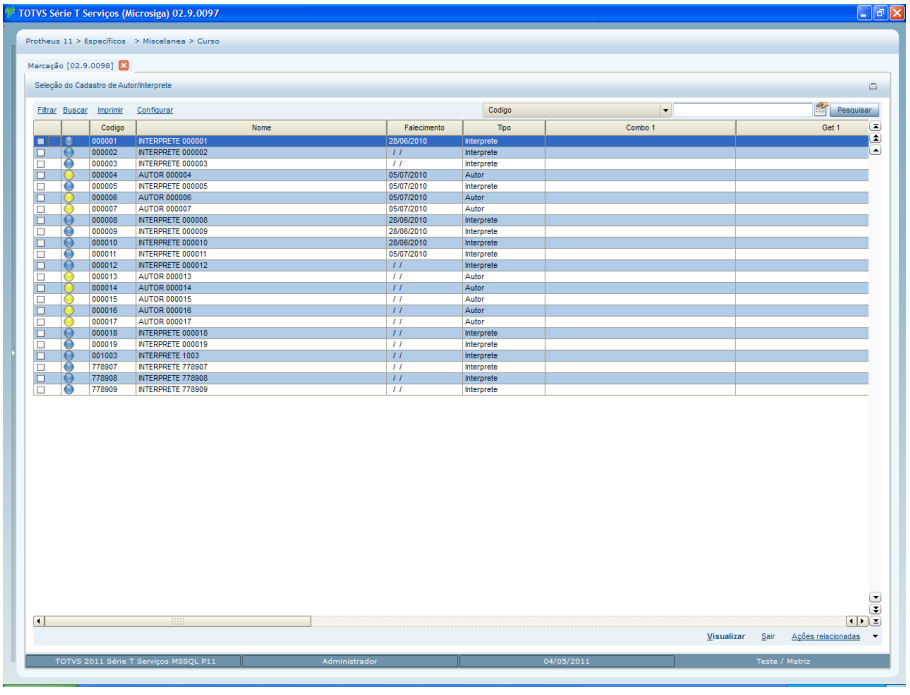
Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
em outra aplicacao Return FWLoadView(
'COMP011_MVC' )

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

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

            Return NIL
```

Formatado: Português (Brasil)

Visualmente tenemos:



14. Múltiples Browses

Como en el uso de la clase ***FWmBrowse*** podemos escribir aplicaciones con más de un objeto de esta clase, es decir, podemos escribir aplicaciones que trabajarán con múltiples ***Browses***.

Podemos por ejemplo desarrollar una aplicación para los pedidos de venta, donde tendremos un ***Browse*** con los encabezados de los ítems, otra con los ítems en la misma pantalla y conforme vamos navegando por los registros del –***Browse*** de encabezado, automáticamente los ítems son actualizados en el otro ***Browse***.

Para esto, crearemos en nuestra aplicación 2 objetos de ***FWMBrowse*** y los relacionamos entre sí. Abajo describimos como hacer esto. Crearemos una aplicación con 3 ***Browses***.

Primero creamos una pantalla ***Dialog*** común, cada uno de los ***Browses*** deben estar ~~anclados~~ fijados en un objeto contenedor, para esto usaremos la función ***FWLayer*** con 2 líneas y en una de esas líneas colocaremos 2 columnas.

Para más detalles de ***FWLayer*** consulte la documentación específica en el TDN[4].

[4] TDN - TOTVS Developer Network portal para desarrolladores de Microsiga Protheus

```
User Function COMP024_MVC()
Local aCoors := FWGetDialogSize( oMainWnd )
Local oPanelUp, oFWLayer, oPanelLeft, oPanelRight, oBrowseUp, oBrowseLeft,
oBrowseRight, oRelacZA4, oRelacZA5



Define MsDialog oDlgPrinc Title 'Multiplos FWmBrowse' From aCoors[1], aCoors[2] To
aCoors[3], aCoors[4] Pixel



//
// Crea contenerdor donde serán colocados los  browses
//
oFWLayer := FWLayer():New()
oFWLayer:Init( oDlgPrinc, .F., .T. )
```

Formatado: Espanhol (México)

```
//
// Painel Inferior
//


oFWLayer:AddLine( 'DOWN', 50, .F. )// Crea una "linha" con 50% de la pantalla
oFWLayer:AddCollumn( 'LEFT' ,50, .T., 'DOWN' )// En la "linha" creada ,se crea una columna con el 50% de su tamaño
oFWLayer:AddCollumn( 'RIGHT',50, .T., 'DOWN' )  // En la "linha" creada, se crea una columna con el 50% de su tamaño
oPanelLeft := oFWLayer:GetColPanel( 'LEFT' , 'DOWN' ) // Pego el objeto del lado izquierdo
oPanelRight := oFWLayer:GetColPanel( 'RIGHT', 'DOWN' ) // Pego el objeto den el pedazo derecho
```

~~Realizado~~ Después de esto creamos 3 **Browses** ~~conforme a~~según lo descrito en el capítulo 3.Aplicaciones con Browses (***FWMBrowse***).

Este es el **1er *Browse*.**

```
//
//      FWmBrowse      Superior      Albuns
//
oBrowseUp:= FWmBrowse():New()
oBrowseUp:SetOwner( oPanelUp )                 // Aqui se associa o browse ao
                                          //componente de tela
oBrowseUp:SetDescription( "Albuns" )
oBrowseUp:SetAlias( 'ZA3' )
oBrowseUp:SetMenuDef( 'COMP024_MVC' )          // Define de onde virao os
                                        // botoes deste browse
oBrowseUp:SetProfileID( '1' )                  // identificador (ID) para o
Browse
oBrowseUp:ForceQuitButton()                    // Força exibição do botão
// Sair
oBrowseUp:Activate()
```

Note, 2 métodos definidos en este ***Browse***: ***SetProfileID*** y ***ForceQuitButton***

El método ***SetProfileID*** define un identificador (*ID*) para el ***Browse***, esto es necesario ya que tenemos más de un ***Browse*** en la aplicación.

El método ***ForceQuitButton*** hace que el botón ***Salir*** se muestre en las opciones del ***Browse***. Como habrá más de un ***Browse*** el botón ***Salir*** no será colocado automáticamente en ninguno de ellos, este método hace que aparezca en el ***Browse***.

Note también, que utilizamos el método ***SetMenuDef*** para definir ~~de cual fuente deberá ser utilizado para a~~el fuente que se utilizará para obtener el ***MenuDef***. Cuando no utilizamos el ***SetMenuDef*** automáticamente el Browse busca el propio fuente donde se encuentra el ***Menudef*** ~~a ser usado~~que se utilizará.

Estos son el **2do** y el **3er *Browses***:

```
oBrowseLeft:= FWMBrowse():New()
oBrowseLeft:SetOwner( oPanelLeft )
oBrowseLeft:SetDescription( 'Musicas' )
 oBrowseLeft:SetMenuDef( '' ) // Referencia vazia para que nao
                              // exiba nenhum botão
oBrowseLeft:DisableDetails()
oBrowseLeft:SetAlias( 'ZA4' )
oBrowseLeft:SetProfileID( '2' )
```


Formatado: Português (Brasil)
Formatado: Português (Brasil)
Formatado: Português (Brasil)

```
oBrowseLeft:Activate()


oBrowseRight:= FWMBrowse():New()
oBrowseRight:SetOwner( oPanelRight )
oBrowseRight:SetDescription( 'Autores/Interpretes' )
oBrowseRight:SetMenuDef( '' ) // Referencia vazia para que nao funcao
                              // exiba nenhum botao
oBrowseRight:DisableDetails()
oBrowseRight:SetAlias( 'ZA5' )
oBrowseRight:SetProfileID( '3' )
oBrowseRight:Activate()
```

Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Português (Brasil)
Formatado: Inglês (Estados Unidos)

Note que en ~~estoy~~ estos ***Browses*** utilizamos el método ***SetMenuDef*** con una referencia vacía, como queremos que el ***Browse*** principal tenga botones de acción, si no definimos el ***SetMenuDef***, automáticamente, el ***Browse*** busca el propio fuente donde ~~-~~se encuentra y con la referencia vacía ~~no son mostrados los botones~~los botones no aparecen.

Ahora que definimos los ***Browses*** necesitamos relacionarlos entre ellos, para que al efectuar el movimiento en uno, automáticamente, los otros sean actualizados.

Para crear la relación utiliza~~re~~mos la clase ***FWBrwRelation***. ~~Similarmente~~ De manera similar a la relación que se hace entre ~~entidades~~ entes en el modelo de datos (*Model*), para ello es necesario ~~decir~~ informar ~~cuáles son las llaves~~las claves de la relación ~~del~~ que hay del Browse ~~**hijo**~~ **secundario** ~~para el~~al browse ~~**padre**~~**principal**.

~~Instanciaremos~~ Creamos el ***FWBrwRelation*** y ~~usaremos~~ utilizamos su método AddRelation. La sintaxis de este método del FWBrwRelation es:

AddRelation(<oBrowsePai>, <oBrowseFilho>, <~~Vetor~~ Vector con los campos de ~~relacion~~relación>) Como tenemos 3 Browses tendremos 2 relaciones:

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


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

~~Sigue~~ Vea un ejemplo completo de aplicación con múltiples **Browses**:

```
User Function COMP024_MVC()
```

```
Local aCoors                := FWGetDialogSize( oMainWnd )
Local oPanelUp, oFWLayer, oPanelLeft, oPanelRight, oBrowseUp,
oBrowseLeft, oBrowseRight, oRelacZA4, oRelacZA5

Private oDlgPrinc

Define MsDialog oDlgPrinc Title 'Multiplos FWmBrowse' From aCoors[1],
aCoors[2] To aCoors[3], aCoors[4] Pixel

//
// Crear el contenedor donde serán colocados los browses
//
oFWLayer := FWLayer():New()
oFWLayer:Init( oDlgPrinc, .F., .T. )

//
// Define Panel Superior
//
oFWLayer:AddLine( 'UP', 50, .F. )
// Cria uma "linha" com 50% da tela
oFWLayer:AddCollumn( 'ALL', 100, .T., 'UP' )
// Na "linha" criada eu crio uma coluna com 100% da tamanho dela oPanelUp :=
oFWLayer:GetColPanel( 'ALL', 'UP' )
// Pego o objeto desse pedaço do container

//
// Panel Inferior
//
oFWLayer:AddLine( 'DOWN', 50, .F. )
// Cria uma "linha" com 50% da tela
oFWLayer:AddCollumn( 'LEFT' ,                50, .T., 'DOWN' )
// Na "linha" criada eu crio uma coluna com 50% da tamanho dela
oFWLayer:AddCollumn( 'RIGHT', 50, .T., 'DOWN' )
// Na "linha" criada eu crio uma coluna com 50% da tamanho dela
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

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


//
// FWmBrowse Superior Albuns
//
oBrowseUp:= FWmBrowse():New()
oBrowseUp:SetOwner( oPanelUp )
// Aqui se associa o browse ao componente de tela oBrowseUp:SetDescription(
"Albuns" )
oBrowseUp:SetAlias( 'ZA3' )
oBrowseUp:SetMenuDef( 'COMP024_MVC' )
// Define de onde virao os botoes deste browse oBrowseUp:SetProfileID( '1' )
oBrowseUp:ForceQuitButton()
oBrowseUp:Activate()


//
// Lado Esquerdo Musicas
//
oBrowseLeft:= FWMBrowse():New()
oBrowseLeft:SetOwner( oPanelLeft )
oBrowseLeft:SetDescription( 'Musicas' )
oBrowseLeft:SetMenuDef( '' )
// Referencia vazia para que nao exiba nenhum botao oBrowseLeft:DisableDetails()
oBrowseLeft:SetAlias( 'ZA4' )
oBrowseLeft:SetProfileID( '2' )
oBrowseLeft:Activate()


//
// Lado Direcho Autores/Interpretes
//
oBrowseRight:= FWMBrowse():New()
oBrowseRight:SetOwner( oPanelRight )
oBrowseRight:SetDescription( 'Autores/Interpretes' ) oBrowseRight:SetMenuDef( '' )
// Referencia vazia para que nao exiba nenhum botao
oBrowseRight:DisableDetails()
oBrowseRight:SetAlias( 'ZA5' )oBrowseRight:SetProfileID( '3' )
oBrowseRight:Activate()
```

Formatado: Inglês (Estados Unidos)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Inglês (Estados Unidos)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

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

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

```
//
// Relacionamento entre os Paineis
/
oRelacZA4:= FWBrwRelation():New()
oRelacZA4:AddRelation( oBrowseUp, oBrowseLeft , { { 'ZA4_FILIAL', 'xFilial( "ZA4" )' }, {'ZA4_ALBUM' , 'ZA3_ALBUM'} } )

oRelacZA4:Activate()

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

oRelacZA5:Activate()

Activate MsDialog oDlgPrinc Center

Return NIL

//---------------------------------------------------------------------
Static Function MenuDef()
Return FWLoadMenuDef( 'COMP023_MVC' )

//---------------------------------------------------------------------
Static Function ModelDef()
// Utilizamos um model que ja existe
Return FWLoadModel( 'COMP023_MVC' )

//---------------------------------------------------------------------
Static Function ViewDef()
// Utilizamos uma View que ja existe
Return FWLoadView( 'COMP023_MVC' )
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

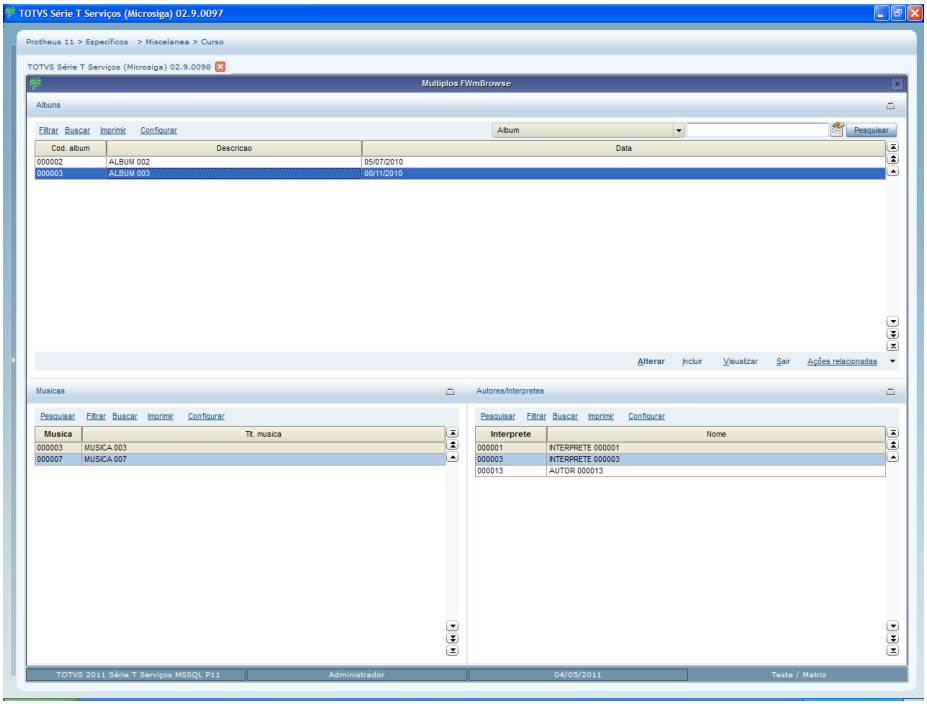
Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Visualmente tenemos:



15. Rutina automática

Cuando una aplicación ~~es desarrolladase~~ desarrolla utilizando el concepto de *MVC* y sus clases, se puede ~~hacer el uso de~~utilizar su modelo de datos en otras aplicaciones, de manera similar a lo que sería una **rutina automática** en desarrollo tradicional.

No existe más el uso de la función ***MSExecAuto***. La idea básica es instanciar (crear) el modelo de datos (*Model*) que se desea, ~~asignar~~ atribuir los valores a ~~él~~ éste y hacer la validación.

Para entenderlo mejor, usaremos ~~de~~ como ejemplo ~~del~~ el fuente que se muestra a continuación, donde se hace en *MVC* lo que sería una **rutina automática** para importación de un cadastro simple.

Observe los comentários.

```
//-------------------------------------------------------------------
// Rotina principal de Importación
//-------------------------------------------------------------------
User Function COMP031_MVC()
```

Formatado: Espanhol (México)

```
Local     aSay := {}
Local     aButton := {}
Local     nOpc := 0
Local     Titulo := 'IMPORTACAO DE COMPOSITORES'
Local     cDesc1 := 'Esta rotina fará a importação de compositores/interpretes'
Local     cDesc2 := 'conforme layout.'
Local     cDesc3 := ''
Local     lOk := .T.
aAdd( aSay, cDesc1 )
aAdd( aSay, cDesc2 )
aAdd( aSay, cDesc3 )

aAdd( aButton, { 1, .T., { || nOpc := 1, FechaBatch() } } )
aAdd( aButton, { 2, .T., { || FechaBatch() } } )
FormBatch( Titulo, aSay, aButton )


If nOpc == 1
Processa( { || lOk := Runproc() },'Aguarde','Processando...',.F.)


If lOk
 ApMsgInfo( 'Processamento terminado com sucesso.', 'ATENÇÃO' )
 Else
 ApMsgStop( 'Processamento realizado com problemas.', 'ATENÇÃO' )
 EndIf
EndIf


Return NIL
//---------------------------------------------------------------------
// Rotina Auxiliar de Importação
//---------------------------------------------------------------------
Static Function Runproc()
Local lRet := .T.
Local aCampos := {}
// Creamos un vector con los datos para facilitar la manupulación de los datos
aCampos := {}
aAdd( aCampos, { 'ZA0_CODIGO', '000100'} )
aAdd( aCampos, { 'ZA0_NOME', 'Vila Lobos'} )
aAdd( aCampos, { 'ZA0_NOTAS' , 'Observações...' } )
aAdd( aCampos, { 'ZA0_TIPO', 'C'} )
If !Import( 'ZA0', aCampos )
 lRet := .F.
EndIf

// Importamos otro registro
aCampos := {}
aAdd( aCampos, { 'ZA0_CODIGO', '000102'} )
aAdd( aCampos, { 'ZA0_NOME', 'Tom Jobim' } )
aAdd( aCampos, { 'ZA0_NOTAS' , 'Observações...' } ) aAdd( aCampos, { 'ZA0_TIPO' ,
'C' } )

If !Import( 'ZA0', aCampos )
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Inglês (Estados Unidos)

Formatado: Português (Brasil)

Formatado: Espanhol (Espanha - tradicional)

Formatado: Português (Brasil)

```
 lRet := .F.
EndIf
// Importamos otro registro
aCampos := {}
aAdd( aCampos, { 'ZA0_CODIGO', '000104'  } )
aAdd( aCampos, { 'ZA0_NOME' , 'Emilio Santiago' } ) aAdd( aCampos, { 'ZA0_NOTAS',
'Observações...' } ) aAdd( aCampos, { 'ZA0_TIPO' , 'I' } )


If !Import( 'ZA0', aCampos )
lRet := .F.
EndIf

Return lRet

//-------------------------------------------------------------------
// Importación de los datos
//-------------------------------------------------------------------
Static Function Import( cAlias, aCampos )
Local   oModel, oAux, oStruct
Local   nI := 0
Local   nPos:= 0
Local   lRet:= .T.
Local   aAux:= {}

dbSelectArea( cAlias )

dbSetOrder( 1 )

// Aqui ocurre el instânciamento del modelo de datos (Model)
// En este ejemplo instanciamos el modelos de datos del fuente COMP011_MVC
// es la rutina de mantenimiento de compositores/interpretes

oModel := FWLoadModel( 'COMP011_MVC' )
// Tenemos que definir cual es la operación que se desea: 3 - Inclusão / 4 -
Alteração / 5 - Exclusão

oModel:SetOperation( 3 )

// Antes de asignar los valores de los campos tenemos que activar el modelo

oModel:Activate()

// Instanciamos solo las referencias de los datos

oAux := oModel:GetModel( cAlias + 'MASTER' )

// Obtenemos la estructura de los datos

oStruct := oAux:GetStruct()
aAux := oStruct:GetFields()

For nI := 1 To Len( aCampos )
```

Formatado: Português (Brasil)

Formatado: Espanhol (Espanha - tradicional)

Formatado: Inglês (Estados Unidos)

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

```
// Verifica si los campos pasado existen en la estructura del modelo

If ( nPos := aScan(aAux,{|x| AllTrim( x[3] )== AllTrim(aCampos[nI][1]) } ) ) > 0

// Se hace la asignación del modelos de datos al campo del model

If !( lAux := oModel:SetValue( cAlias + 'MASTER', aCampos[nI][1], aCampos[nI][2]
) )

// en caso de aignación  no se pueda realizar por algún motivo (validación, por
ejemplo)
// el método SetValue retorna .F.

lRet := .F.
Exit
EndIf
EndIf
Next nI

If lRet

// Hace la validación de los datos, note que diferentemente de las tradicionales
// "rutinas automáticas"
// en este momento los datos no son grabados, si no solamente validaos.

If ( lRet := oModel:VldData() )

// Si los datos fueran validados hace la grabación efectiva de los datos (commit)

oModel:CommitData()
EndIf
EndIf

If !lRet

//Si los datos no fueran validos obtenemos la descripción del error para genear LOG
//o mensaje de aviso

aErro := oModel:GetErrorMessage()

// La estructura del vector con erro es:
//[1] identificador (ID) del formulario de origen
//[2] identificador (ID) del campo de origen
//[3] identificador (ID) del formulario de error
//[4] identificador (ID) del campo de error
//[5] identificador (ID) del erroe
//[6] mensaje de error
//[7] mensaje de la solución
//[8] Valor asigando
//[9] Valor anterior
AutoGrLog( "Id do formulário de origem:" + ' [' + AllToChar( aErro[1] ) + ']' )
AutoGrLog( "Id do campo de origem: " + ' [' + AllToChar( aErro[2]) + ']' )
AutoGrLog( "Id do formulário de erro:   " + ' [' + AllToChar( aErro[3]) + ']' )
AutoGrLog( "Id do campo de erro: " + ' [' + AllToChar( aErro[4]) + ']' )
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
AutoGrLog( "Id do erro: " + ' [' + AllToChar( aErro[5]) + ']' )
AutoGrLog( "Mensagem do erro: " + ' [' + AllToChar( aErro[6]) + ']' )
AutoGrLog( "Mensagem da solução: " + ' [' + AllToChar( aErro[7]) + ']' )
AutoGrLog( "Valor atribuído: " + ' [' + AllToChar( aErro[8]) + ']' )
AutoGrLog( "Valor anterior:" + ' [' + AllToChar( aErro[9]) + ']' )
MostraErro()
EndIf
// Desactivamos el Model
oModel:DeActivate()

Return lRet
```

En este otro ejemplo, temos la importación ~~para~~ a un modelo de datos donde hay una estructura de ***Master-Detail*** (~~***Padre-Hijo***~~***Principal-Secundario***). También lo que haremos es instanciar el modelo de datos (*Model*) que deseamos, ~~asignar~~ atribuir los valores ~~a el~~al mismo y hacer la validación, ~~solo~~ sólo que haremos esto para ~~las dos entidades~~los dos entes.

Observe los comentarios:

```
//-------------------------------------------------------------------
// Rutina principal de Importación
//-------------------------------------------------------------------
User Function COMP032_MVC()
Local     aSay := {}
Local     aButton := {}
Local     nOpc := 0
Local     Titulo := 'IMPORTACAO DE MUSICAS'
Local     cDesc1 := 'Esta rotina fará a importação de musicas'
Local     cDesc2 := 'conforme layout.'
Local     cDesc3 := ''
Local     lOk := .T.

aAdd( aSay, cDesc1 )
aAdd( aSay, cDesc2 )
aAdd( aSay, cDesc3 )

aAdd( aButton, { 1, .T., { || nOpc := 1, FechaBatch() } } )
aAdd( aButton, { 2, .T., { || FechaBatch() } } )

FormBatch( Titulo, aSay, aButton )
If nOpc == 1
Processa( { || lOk := Runproc() },'Aguarde','Processando...',.F.)
 If lOk
ApMsgInfo( 'Processamento terminado com sucesso.', 'ATENÇÃO' )
Else
ApMsgStop( 'Processamento realizado com problemas.', 'ATENÇÃO' )
EndIf

EndIf
Return NIL

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

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
// Rutina auxiliar de Importación
//---------------------------------------------------------------------
Static Function Runproc()
Local lRet := .T.
Local aCposCab := {}
Local aCposDet := {}
Local aAux := {}

//Creamos un vector con los datos del encabezado y otro para los Ítems,
//para facilitar el mantenimiento de los datos

aCposCab := {}
aCposDet := {}
aAdd( aCposCab, { 'ZA1_TITULO' , 'LA, LA, LA,' } ) aAdd( aCposCab, { 'ZA1_DATA',
Date() } )


aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '01' } )
aAdd( aAux, { 'ZA2_AUTOR', '000100' } )
aAdd( aCposDet, aAux )


aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '02' } )
aAdd( aAux, { 'ZA2_AUTOR', '000104' } )
aAdd( aCposDet, aAux )
If !Import( 'ZA1', 'ZA2', aCposCab, aCposDet )
 lRet := .F.
EndIf
// Importamos otro conjunto de datos
aCposCab := {}
aCposDet := {}
aAdd( aCposCab, { 'ZA1_TITULO' , 'BLA, BLA, BLA' } ) aAdd( aCposCab, { 'ZA1_DATA',
Date() } )

aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '01' } )


aAdd( aAux, { 'ZA2_AUTOR', '000102' } )
aAdd( aCposDet, aAux )
aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '02' } )
aAdd( aAux, { 'ZA2_AUTOR', '000104' } )
aAdd( aCposDet, aAux )
If !Import( 'ZA1', 'ZA2', aCposCab, aCposDet )
 lRet := .F.
EndIf

// Importamos otro conjunto de datos
aCposCab := {}
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
aCposDet := {}
aAdd( aCposCab, { 'ZA1_TITULO' , 'ZAP, ZAP, ZAP' } ) aAdd( aCposCab, { 'ZA1_DATA',
Date() } )

aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '01' } )
aAdd( aAux, { 'ZA2_AUTOR', '000100' } )
aAdd( aCposDet, aAux )


aAux := {}
aAdd( aAux, { 'ZA2_ITEM' , '02' } )
aAdd( aAux, { 'ZA2_AUTOR', '000102' } )
aAdd( aCposDet, aAux )
If !Import( 'ZA1', 'ZA2', aCposCab, aCposDet )
 lRet := .F.
EndIf

Return lRet

//-------------------------------------------------------------------
// Importaciacón de los datos
//-------------------------------------------------------------------
Static Function Import(   cMaster, cDetail, aCpoMaster, aCpoDetail )
Local   oModel, oAux, oStruct
Local   nI := 0
Local   nJ := 0

Local   nPos:= 0
Local   lRet:= .T.
Local   aAux:= {}
Local   aC := {}
Local   aH := {}
Local   nItErro:= 0
Local   lAux:= .T.
dbSelectArea( cDetail )
dbSetOrder( 1 )
dbSelectArea( cMaster )
dbSetOrder( 1 )
// Aquí ocurre el instanciamiento del modelo de datos (Model)
// En este ejemplo instanciamos el modelo de datos del fuente COMP022_MVC
// que es a rutina de mantenimiento de música
oModel := FWLoadModel( 'COMP022_MVC' )
// Tenemos que definir cuál es la operación : 3 - Incluir / 4 - Alterar / 5 -
Excluir
oModel:SetOperation( 3 )
// Antes de asignar los valores de los campos tenemos que activar el modelo
oModel:Activate()
// Instanciamos solo la parte del modelo referente a los dados de encabezado
oAux := oModel:GetModel( cMaster + 'MASTER' )
// Obtenemos la estructura de dados del encabezado
oStruct := oAux:GetStruct()
```

Formatado: Espanhol (Espanha - tradicional)

Formatado: Espanhol (Espanha - tradicional)

Formatado: Espanhol (Espanha - tradicional)

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

```
aAux := oStruct:GetFields()
If lRet
 For nI := 1 To Len( aCpoMaster )
  // Verifica si los campos pasados existen en la estructura del encabezado
  If(  nPos :=   aScan(  aAux,{ |x|   AllTrim(  x[3] ) ==AllTrim(
   aCpoMaster[nI][1] ) } ) ) > 0
   //hace la asignación del dato a los campos del Model del encabezado
   If !( lAux := oModel:SetValue( cMaster + 'MASTER', aCpoMaster[nI][1],
    aCpoMaster[nI][2] ) )
    //  En  caso  de  que  la  asignación  no  pueda  ser  realizada,por  algún
motivo(validación, por ejemplo)
    // el método SetValue retorna .F.
    lRet := .F.
    Exit
   EndIf
  EndIf
 Next
EndIf
If lRet
// Instanciamos solo la parte del modelo referente a los dados de ítem
oAux := oModel:GetModel( cDetail + 'DETAIL' )
// Obtenemos la estructura de dados del ítem
oStruct := oAux:GetStruct()
aAux := oStruct:GetFields()
nItErro := 0
For nI := 1 To Len( aCpoDetail )
 // Incluimos una línea nueva
 //  ATENCIÓN:  Loas  ítems  son  creados en una estructura de grid(FORMGRID),
 //por tanto ya fue creado una primera línea
 //en blanco  automáticamente,  de esta  forma  comenzamos a  insertar  nuevas
líneas a partir de 2ª vez
 If nI > 1
  // Incluimos una nueva línea de ítem
  If ( nItErro := oAux:AddLine() ) <> nI
   // Se por algún motivo el método AddLine() no incluye la línea,
   // este retornara la cantidad de líneas que ya
   // existen en el grid. y si la incluye, retornara la cantidad más 1
   lRet := .F.
   Exit
  EndIf
 EndIf


 For nJ := 1 To Len( aCpoDetail[nI] )
  // Verifica si los campos pasados existen en la estructura de ítem
  If ( nPos := aScan( aAux, { |x| AllTrim( x[3] ) ==  AllTrim(
   aCpoDetail[nI][nJ][1] ) } ) ) > 0
   If !( lAux := oModel:SetValue( cDetail + 'DETAIL',
    aCpoDetail[nI][nJ][1], aCpoDetail[nI][nJ][2] ) )
    //  En  caso  de  que  la  asignación  no  pueda  ser  realizada,por  algún
motivo(validación, por ejemplo)
    // el método SetValue retorna .F.
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Inglês (Estados Unidos)

```
    lRet := .F.
    nItErro := nI
    Exit
   EndIf
  EndIf
 Next
 If !lRet
 Exit
 EndIf
Next
EndIf


If lRet
 // Se realiza la  validación de los datos, note que a diferencia de las
tradicionales "rutinas automáticas"
 // en este momento los datos no son grabados, son solamente validados.
 If ( lRet := oModel:VldData() )
  // Si los datos fueran validos se realiza la grabación de los datos (Commit)
  oModel:CommitData()
 EndIf
EndIf

If !lRet
 // Si los datos no fueran validos obtenemos la descripción de error para generar
 // LOG o mensaje de aviso
 aErro := oModel:GetErrorMessage()
 // La estructura del vector con los errores es:
 // [1] identificador (ID) del formulario de origen
 // [2] identificador (ID) del campo de origen
 // [3] identificador (ID  del formulario de error
 // [4] identificador (ID) del campo de error
 // [5] identificador (ID) del error
 // [6] mensaje de error
 // [7] mensaje de solución
 // [8] Valor atribuido
 // [9] Valor anterior
 AutoGrLog( "Id do formulário de origem:" + ' [' + AllToChar( aErro[1] ) + ']' )
 AutoGrLog( "Id do campo de origem: " + ' [' + AllToChar( aErro[2]) + ']' )
 AutoGrLog( "Id do formulário de erro:   " + ' [' + AllToChar( aErro[3]) + ']' )
 AutoGrLog( "Id do campo de erro: " + ' [' + AllToChar( aErro[4]) + ']' )
 AutoGrLog( "Id do erro: " + ' [' + AllToChar( aErro[5]) + ']' )
 AutoGrLog( "Mensagem do erro: " + ' [' + AllToChar( aErro[6]) + ']' )
 AutoGrLog( "Mensagem da solução: " + ' [' + AllToChar( aErro[7]) + ']' )
 AutoGrLog( "Valor atribuído: " + ' [' + AllToChar( aErro[8]) + ']' )
 AutoGrLog( "Valor anterior:" + ' [' + AllToChar( aErro[9]) + ']' )
 If nItErro > 0
  AutoGrLog(  "Erro  no  Item: " +  ' ['+  AllTrim(  AllToChar(nItErro) ) + ']' )
 EndIf
 MostraErro()
EndIf
```

Formatado: Inglês (Estados Unidos)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

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

```
// Desactivamos o Model
oModel:DeActivate()
Return lRet
```

Una situación que podrá ser encontrada, en los casos en que se esté convirtiendo una aplicación que ya existe para la estructura de *MVC*, es el hecho de que la aplicación ya puede estar preparada para trabajar como rutina automática y por consecuencia pueden existir otras aplicaciones que ya utilicen esa rutina automática.

La función ***FWMVCRotAuto*** fue creada para que no sea necesario que estas aplicaciones, que hoy se usan en la llamada de la rutina estándar, cambien su forma de trabajar, ya que la aplicación fue convertida ~~para~~ a *MVC*.

La función utiliza los parámetros pasados en el formato anterior de la rutina automática (***MSEXECAUTO***) y ~~hace el instanciamiento del~~crea el *model*, la ~~asignación~~ atribución de valores y la validación en el formato *MVC*, garantizando los programas heredados.

Formatado: Fonte: Itálico

Sintaxis:

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

Donde:

oModel Objeto con el modelo de formulario de datos~~;~~

cAlias Alias del Browse principal~~;~~

nOpcAuto Código de identificación del tipo de procesamiento de la rutina automática;

[3] Inclusión

[4] ~~Alteración~~Modificación

[5] ~~Exclusión~~Borrado

aAuto Array con los datos de la rutina automática en la siguiente estructura~~;~~

[n][1] Código del formulario del Modelo que tendrá una ~~asignación~~atribución~~;~~

[n][2] Array estándar de los datos del EnchAuto y GetDAuto, ~~conforme a~~según la documentación anterior~~;~~

lSeek Indica si el archivo principal debe ser posicionado con base en los datos proporcionados~~;~~

lPos Indica si el nOpc no debe ser calculado con base en la aRotina~~;~~
Así mismo la aplicación en *MVC* que fue convertida podrá trabajar de las dos formas:

- Como rutina automática y o

- Instanciamiento del *model*.

En el ejemplo a continuación tenemos una rutina de ~~cadastro~~ registro donde hay un tratamiento para esto, si los datos ***xRotAuto***, ***nOpcAuto*** son pasados, indica que la aplicación fue llamada por una rutina automática~~, y así mismo~~por eso utilizamos la función ***FWMVCRotAuto***.

~~En c~~Esta construcción no impide que en las otras aplicaciones también se instancie el modelo de datos (*Model*) directamente.

```
Function MATA030_MVC(xRotAuto,nOpcAuto)

Local oMBrowse
If xRotAuto == NIL
  oBrowse := FWMBrowse():New()
  oBrowse:SetAlias('SA1')
  oBrowse:SetDescription("Cadastro de Clientes") oBrowse:Activate()
Else
  aRotina := MenuDef()
  FWMVCRotAuto(ModelDef(),"SA1",nOpcAuto,{{"MATA030_SA1",xRotAuto}})
Endif

Return NIL
```

Formatado: Espanhol (México)

16. Puntos de entrada en MVC

Pontos de entrada son desvíos controlados que se ejecutan durante las aplicaciones.

Si se escribe una aplicación utilizando ~~-~~*MVC*, automáticamente ya estarán disponibles puntos de entrada pre~~-~~definidos.

La ~~ida~~ idea del punto de entrada, para fuentes desarrollados que utilizan el concepto de *MVC* y sus clases, es un poco diferente ~~a~~ de las aplicaciones desarrolladas de ~~forma~~ manera convencional.

En los fuentes convencionales tenemos un **nombre** para cada punto de entrada , por ejemplo, en la rutina ***MATA010*** - ~~Cadastro~~ Archivo de Productos, tenemos los puntos de entrada: ***MT010BRW***, ***MTA010OK***, ***MT010CAN***, etc. En *MVC*, no es de esta ~~forma~~manera.

En *MVC* creamos un único punto de entrada y este es llamado en varios momentos dentro de la aplicación desarrollada.

Este punto de entrada único debe ser una ***User Function*** y debe tener como nombre el identificador (*ID*) del modelo de datos (*Model*) del fuente. Tomemos por ejemplo un fuente del Modulo Jurídico: ***JURA001***. En ~~este~~ dicho fuente el identificador (*ID*) del modelo de datos (definido en la función ***ModelDef***) es también ***JURA001***, por lo tanto si se escribe un punto de entrada de esta aplicación, haríamos:

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

Formatado: Espanhol (Espanha - tradicional)

```
Return xRet
```

El punto de entrada creado recibe vía parámetro (***PARAMIXB***) un vector con informaciones referentes a la aplicación. Estos parámetros varían para cada situación, en común todos, en los 3 primeros elementos –son listados a continuación, y en el siguiente cuadro existe la relación de parámetros para cada *ID*:

Posiciones del array de parámetros comunes a todos los *IDs*:

POS.	TIPO	DESCRIPCIÓN
1	O	Objeto del formulario o del modelo, conforme al caso
2	C	ID del local de ejecución del punto de entrada
3	C	ID del formulario

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

Como ya se ~~mencionó~~mencionó, el punto de entrada es llamado en varios momentos dentro de la aplicación, en la 2da posición de la estructura del vector es pasado un identificador (*ID*) que identifica cual es ese momento. Esta posición puede tener como contenido:

ID DEL PUNTODE ENTRADA	MOMENTO DE EJECUCIÓN DEL PUNTO DE ENTRADA
MODELPRE	Antes de ~~la alteración de~~modificar cualquier campo del modelo. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: Requiriere un retorno lógico.
MODELPOS	En la validación total del modelo. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: Requiere un retorno lógico.
FORMPRE	Antes de ~~la alteración de~~modificar cualquier campo del formulario. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: Requiere un retorno lógico.
FORMPOS	En la validación total del formulario. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso.

	2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: Requiere un retorno lógico.
FORMLINEPRE	Antes de ~~la alteración de~~modificar la línea del formulario FWFORMGRID. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme al caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. 4 N Número de línea del FWFORMGRID. 5 C Acción de la FWFORMGRID. 6 C Id del campo. Retorno: Requiere un retorno lógico.
FORMLINEPOS	En la validación total de la línea del formulario FWFORMGRID. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. 4 N Número de la línea del FWFORMGRID. Retorno: Requiere un retorno lógico.
MODELCOMMITTTS	Después de la grabación total del modelo y dentro de la transacción. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno:

	No espera retorno.
MODELCOMMITNTTS	Después de la grabación total del modelo y fuera de la transacción. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme al caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: No espera retorno.
FORMCOMMITTTSPRE	Antes de la grabación de la tabla del formulario. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. 4 L Si es .T. (verdadero) indica si es nuevo registro (Inclusión) si es .F. (falso) el registro ya existe (~~Alteración~~ Modificación / ~~Exclusión~~Borrado). Retorno: No espera retorno.
FORMCOMMITTTSPOS	Después de la grabación de la tabla del formulario. Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. 4 L Si es .T. (verdadero) indica si es nuevo registro (Inclusión) si es .F. (falso) indica si el registro ya existe (~~Alteración~~ Modificación / ~~Exclusión~~Borrado). Retorno: No espera retorno.
FORMCANCEL	El botón de ~~cancelar~~anular. Parámetros Recibidos:

	1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: Requiere un retorno lógico.
MODELVLDACTIVE	Activación del modelo Parámetros Recibidos: 1 O Objeto del formulario ~~ó~~ o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: Requiere un retorno lógico.
BUTTONBAR	Incluir botones en el ControlBar. Para crear los botones se debe retornar un array bidimensional con la siguiente estructura de cada ítem: 1 C Titulo para el botón. 2 C Nombre del Bitmap para mostrar. 3 B CodeBlock a ~~ser ejecutado~~ejecutar. 4 C ToolTip (Opcional). Parámetros Recibidos: 1 O Objeto del formulario o del modelo, conforme el caso. 2 C ID del local de ejecución del punto de entrada. 3 C ID del formulario. Retorno: Array con la estructura ~~pre-definida~~predefinida.

Observaciones:

- Cuando el modelo de datos tiene varios componentes (por ejemplo, de *grid*), la 3ra posición del vector traerá el identificador (ID) de este componente~~;~~.

- Cuando el tipo de retorno de un determinado tiempo de ejecución no es pasado ~~ó~~ o es pasado con el tipo equivocado se mostrará un mensaje en la consola avisando sobre esto. Todos los ID~~s~~ que esperan retorno deben ser tratados en el punto de entrada.

Importante:

- Al escribir una ***User Function*** en *MVC*, tenga cuidado al ~~asignar~~ atribuir el Identificador (ID) del modelo de datos (*Model*), porque no podrá tener el mismo nombre que el fuente (*PRW*). Si el fuente tiene el nombre ***FONT001***, el identificador (*ID*) del modelo de datos (*Model*) no podrá ser también ***FONT001***, por ~~lo que no será~~eso no es posible crear otra ***User Function*** con el nombre ~~de~~ ***FONT001*** (*ID* del modelo de datos) para los puntos de entrada.

Ejemplo:

```
User Function JURA001()

Local aParam := PARAMIXB
Local xRet := .T.
Local oObj := ''
Local cIdPonto := ''
Local cIdModel := ''
Local lIsGrid := .F.
Local nLinha := 0
Local nQtdLinhas := 0
Local cMsg := ''

If aParam <> NIL

 oObj := aParam[1]
 cIdPonto := aParam[2]
 cIdModel := aParam[3]
 lIsGrid := ( Len( aParam ) > 3 )

 If lIsGrid

  nQtdLinhas := oObj:GetQtdLine()
  nLinha := oObj:nLine

 EndIf

 If cIdPonto == 'MODELPOS'

  cMsg := 'Chamada na validação total do modelo (MODELPOS).' + CRLF
  cMsg += 'ID ' + cIdModel + CRLF

  If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )
   Help( ,, 'Help',, 'O MODELPOS retornou .F.', 1, 0 )

  EndIf
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
ElseIf cIdPonto == 'FORMPOS'

 cMsg := 'Chamada na validação total do formulário (FORMPOS).' + CRLF
 cMsg += 'ID ' + cIdModel + CRLF

 If cClasse == 'FWFORMGRID'

  cMsg += 'É um FORMGRID com ' + Alltrim( Str( nQtdLinhas ) ) + ;
  ' linha(s).' + CRLF
  cMsg += 'Posicionado na linha ' + Alltrim( Str( nLinha ) ) + CRLF

 ElseIf cClasse == 'FWFORMFIELD'

  cMsg += 'É um FORMFIELD' + CRLF

 EndIf

 If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )

  Help( ,, 'Help',, 'O FORMPOS retornou .F.', 1, 0 )

 EndIf

ElseIf cIdPonto == 'FORMLINEPRE'

 If aParam[5] == 'DELETE'

  cMsg := 'Chamada na pré validação da linha do formulário (FORMLINEPRE).' +
CRLF
  cMsg += 'Onde esta se tentando deletar uma linha' + CRLF
  cMsg += 'É um FORMGRID com ' + Alltrim( Str( nQtdLinhas ) ) +;
  ' linha(s).' + CRLF
  cMsg += 'Posicionado na linha ' + Alltrim( Str( nLinha ) ) +; CRLF
  cMsg += 'ID ' + cIdModel + CRLF
  If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )
   Help( ,, 'Help',, 'O FORMLINEPRE retornou .F.', 1, 0 )

  EndIf

 EndIf

ElseIf cIdPonto == 'FORMLINEPOS'

 cMsg := 'Chamada na validação da linha do formulário (FORMLINEPOS).' +; CRLF
 cMsg += 'ID ' + cIdModel + CRLF
 cMsg += 'É um FORMGRID com ' + Alltrim( Str( nQtdLinhas ) ) + ;
 ' linha(s).' + CRLF
 cMsg += 'Posicionado na linha ' + Alltrim( Str( nLinha ) ) + CRLF

 If !( xRet := ApMsgYesNo( cMsg + 'Continua ?' ) )

  Help( ,, 'Help',, 'O FORMLINEPOS retornou .F.', 1, 0 )

 EndIf

ElseIf cIdPonto == 'MODELCOMMITTTS'
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
 ApMsgInfo('Chamada apos a gravação total do modelo e dentro da transação
(MODELCOMMITTTS).' + CRLF + 'ID ' + cIdModel )

 ElseIf cIdPonto == 'MODELCOMMITNTTS'

 ApMsgInfo('Chamada apos a gravação total do modelo e fora da transação
(MODELCOMMITNTTS).' + CRLF + 'ID ' + cIdModel)
 //ElseIf cIdPonto == 'FORMCOMMITTTSPRE'

 ElseIf cIdPonto == 'FORMCOMMITTTSPOS'

 ApMsgInfo('Chamada apos a gravação da tabela do formulário
(FORMCOMMITTTSPOS).' + CRLF + 'ID ' + cIdModel)

 ElseIf cIdPonto == 'MODELCANCEL'

  cMsg := 'Chamada no Botão Cancelar (MODELCANCEL).' + CRLF + 'Deseja
Realmente Sair ?'

  If !( xRet := ApMsgYesNo( cMsg ) )

   Help( ,, 'Help',, 'O MODELCANCEL retornou .F.', 1, 0 )

 EndIf

 ElseIf cIdPonto == 'MODELVLDACTIVE'

  cMsg := 'Chamada na validação da ativação do Model.' + CRLF + ;
  'Continua ?'

  If !( xRet := ApMsgYesNo( cMsg ) )

   Help( ,, 'Help',, 'O MODELVLDACTIVE retornou .F.', 1, 0 )

 EndIf

 ElseIf cIdPonto == 'BUTTONBAR'

  ApMsgInfo('Adicionando Botão na Barra de Botões (BUTTONBAR).' + CRLF + 'ID '
+ cIdModel )
  xRet := { {'Salvar', 'SALVAR', { || Alert( 'Salvou' ) }, 'Este botão Salva'
} }

 EndIf

EndIf

Return xRet
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

17.Web Services para MVC

Al desarrollar una aplicación en *MVC*, estará disponible ~~un~~ una *Web Service* para ~~el recibimiento~~recibir ~~de~~ los datos.

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

Todas las aplicaciones en *MVC* utilizarán el mismo Web Service, independientemente de su estructura o del número de entidades que tenga.

El Web Service que esta disponible para *MVC* es ***FWWSMODEL***.

La idea básica es que se instanciará ~~el~~ la Web Service, se informará que aplicación ~~será utilizada~~se utilizará y se informarán los datos en un formato XML.

17.1 Web Service para modelos de datos que tienen ~~una entidad~~un ente

Veremos cómo construir una aplicación que utilice ~~el~~ la Web Service ***FWWSMODEL*** con un modelo de datos (Model) que tiene ~~solo~~ sólo ~~una entidad~~un ente.

17.2 Cómo ~~I~~instanciar el Client de la Web Service

El instanciamiento será de la siguiente forma:

Instanciamiento del Client ~~del~~ de la Web Service.

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

Definición del URL del ***FWWSMODEL*** en el servidor de la *Web Services*.

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

Definición de la aplicación que será utilizada.

Definimos aquí el nombre del fuente que contiene el ***ModelDef*** que queremos utilizar.

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

17.3 La estructura del XML utilizada

Como se dijo anteriormente los datos serán informados en un XML. La estructura de ~~este~~ esta XML sigue la siguiente jerarquía básica:

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

El tag ***<ID del Model>*** es el identificador (ID) que está definido en el modelo de datos (Model) da la aplicación *MVC*.

Ejemplo:

En la aplicación está definido:

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

En el XML lo gags <***ID del Model***> serán:

```
<COMP011M>


</COMP011M>
```

~~La operación~~Las operaciones que ~~puede ser realizada~~se pueden realizar son: **inclusión (3)**, ~~**alteración**~~ **modificación (4)** ~~ó~~ o ~~**exclusión**~~ **borrado (5)**, esta operación tiene que ser informada en el tag en el atributo *Operation*.

Formatado: Fonte: Itálico

~~Así que si quisiéramos~~De este modo, si queremos hacer una operación de Inclusión ~~tendríamos~~tendremos:

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

Los tags <**ID de Componente**> son *IDs* de los componentes de formularios ~~ó~~ o componente de grid que fueron definidos en el modelo de datos (*Model*) de la aplicación.

Ejemplo:

Si en la aplicación tenemos:

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

En el XML los tags <***ID de Componente***> serán:

```
<ZA0MASTER>


</ZA0MASTER>
```

El tipo de componente (del formulario ~~ó~~ o del grid) también debe ser informado en este tag en el atributo modeltype. Informe ***FIELDS*** para componentes de formularios y ***GRID*** para componentes de grid.

Tendremos esto:

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

Los tags <***ID de Campo***> serán los nombres de los campos de la estructura del componente, sea formulario ó grid.
De esta misma forma, si en la estructura tuviéramos los campos ***ZA0_FILIAL***, ***ZA0_ CODIGO*** y ***ZA0_NOME***, por ejemplo, tendríamos:

```
<ZA0_FILIAL>
```

```
</ZA0_FILIAL>

<ZA0_CODIGO>

</ZA0_CODIGO>

<ZA0_NOME>

</ZA0_NOME>
```

El orden de los campos también deben ser informados en los tags, con el atributo ***order***.

```
<ZA0_FILIAL order="1">

</ZA0_FILIAL>
<ZA0_CODIGO order="2">

</ZA0_CODIGO >
<ZA0_NOME     order="3">

</ZA0_NOME>
```

Cuando el componente es un formulario (***FIELDS***), los datos se deben informar en el tag ***value***.

```
<ZA0_FILIAL order="1">
<value>01</value>
</ZA0_FILIAL>
<ZA0_CODIGO order="2">
     <value>001000</value>
</ZA0_CODIGO >
<ZA0_NOME     order="3">
     <value>Tom Jobim</value>
</ZA0_NOME>

Estrutura completa:

<COMP011M Operation="1">
        <ZA0MASTER modeltype="FIELDS" >
             <ZA0_FILIAL order="1">
                <value>01</value>
             </ZA0_FILIAL>
              <ZA0_CODIGO order="2">
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Inglês (Estados Unidos)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Inglês (Estados Unidos)

```
                    <value>01000</value>
                </ZA0_CODIGO>
                <ZA0_NOME     order="3">
                    <value>Tom Jobim</value>
                </ZA0_NOME>
            </ZA0MASTER>
        </COMP011M >
```

17.4 Obtener la estructura XML de un modelo de datos(GetXMLData)

Podemos obtener la estructura XML que una aplicación en *MVC* espera, para esto se utilizará el método ***GetXMLData*** del Web Service.

Ejemplo:

```
oMVCWS:GetXMLData()
```

El XML esperado será informado en el atributo ***cGetXMLDataResult*** del WS.

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

Utilizando el ejemplo anterior, tendríamos:

```
<?xml version="1.0" encoding="UTF-8"?>
<COMP011M Operation="1"
    <ZA0MASTER modeltype="FIELDS" >
        <ZA0_FILIAL order="1"><value></value></ZA0_FILIAL>
        <ZA0_CODIGO order="2"><value></value></ZA0_CODIGO>
        <ZA0_NOME   order="3"><value></value></ZA0_NOME>
     </ZA0MASTER>
</COMP011M>
```

17.5 Informando los datos XML al Web Service

El XML contiene los datos que deben ser asignados al atributo ***cXML*** del objeto de Web Service.

Ejemplo:

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

17.6 Validando los datos (VldXMLData)

Para enviar los datos al modelo de dados (Model) para que sean validados utilizamos el método ***VldXMLData***.

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

Formatado: Espanhol (Espanha - tradicional)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
)
EndIf
```

En este momento los datos son validados por ***VldXMLData*** , mas no son grabados. Este es un recurso interesante si quisiéramos hacer una simulación, por ejemplo.

17.7 Validando la grabación de los datos (PutXMLData)

La diferencia entre el método ***VldXMLData*** y el método ***PutXMLData***, es que el PutXMLData más allá de presentar los datos al XML al modelo de datos para validación, también hará la grabación de estos datos si la validación ~~fue exitosa~~tuvo éxito.

El resultado es informado en el atributo ***lPutXMLDataResult*** en el caso de que haya algún problema, será descrito en el atributo ***cVldXMLDataResult*** del objeto del Web Service.

```
If oMVCWS:PutXMLData()
If oMVCWS:lPutXMLDataResult
            MsgInfo( 'Informação gravada com sucesso.' )
      Else
            MsgStop( 'Informação não gravada ' + CRLF + WSError() )
      EndIf
Else
      MsgStop( AllTrim(oMVCWS:cVldXMLDataResult) + CRLF + WSError() )
EndIf
```

Formatado: Espanhol (México)
Formatado: Espanhol (México)
Formatado: Espanhol (México)
Formatado: Português (Brasil)

17.8 Obteniendo el esquema XSD de un modelo de datos (GetSchema)

El XML informado antes de la validación de la información por el modelo de datos (*Model*) será validado por el esquema XSD referente al modelo. Esta validación ~~es realizada~~se realiza automáticamente y el XSD ~~es basado~~se basa en la estructura del modelo de datos (*Model*).

Esta validación se refiere a la estructura del XML (*tags, nivel, orden, etc.)* y no a los datos del XML, la validación de los datos es función de la regla de negocio.

Si el desarrollador quiere obtener el esquema XSD que será utilizado, podrá usar el método ***GetSchema***.

Ejemplo:

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


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

El esquema XSD es retornado en el atributo ***cGetSchemaResult*** del objeto del Web Service.

17.9 Ejemplo completo de la Web Service

```
User Function COMPW011()

Local oMVCWS

// Instancia el WebService Genérico para Rutinas en MVC o
MVCWS := WsFwWsModel():New()
// URL donde esta el WebService FWWSModel de Protheus
oMVCWS:_URL := http://127.0.0.1:8080/ws/FWWSMODEL.apw
// Seta Atributos del WebService
oMVCWS:cModelId := 'COMP011_MVC' // Fuennte de donde se utilizara el Model
// Ejemplo de como pegar la descripción del Modelo de Dados
//If oMVCWS:GetDescription()
// MsgInfo( oMVCWS:cGetDescriptionResult )
//Else
// MsgStop( 'Problemas em obter descrição do Model'
//EndIf
// Obtiene la estructura de los datos del Model
If oMVCWS:GetXMLData()
// Retorno dela GetXMLData
cXMLEstrut := oMVCWS:cGetXMLDataResult
// Retorna
//<?xml version="1.0" encoding="UTF-8"?>
//<COMP011M Operation="1" version="1.01">
// <ZA0MASTER modeltype="FIELDS" >+ CRLF + WSError() )
// <ZA0_FILIAL order="1"><value></value></ZA0_FILIAL>
// <ZA0_CODIGO order="2"><value></value></ZA0_CODIGO>
// <ZA0_NOME     order="3"><value></value></ZA0_NOME>
// </ZA0MASTER>
//</COMP011M>

// Obtiene el esquema de datos XML (XSD)

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

Formatado: Espanhol (Espanha - tradicional)

Formatado: Espanhol (México)

Formatado: Português (Brasil)

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

```
EndIf
// Crea el XML

cXML := '<?xml version="1.0" encoding="UTF-8"?>'
cXML += '<COMP011M Operation="1" version="1.01">'
cXML += '<ZA0MASTER modeltype="FIELDS" >'
cXML += '<ZA0_FILIAL order="1"><value>01</value></ZA0_FILIAL>'
cXML += '<ZA0_CODIGO order="2"><value>000100</value></ZA0_CODIGO>'
cXML += '<ZA0_NOME     order="3"><value>Tom Jobim</value></ZA0_NOME>'
cXML += '</ZA0MASTER>'
cXML += '</COMP011M>'

// Joga o XML para o atributo do WebService
oMVCWS:cModelXML := cXML
// Valida y Graba los dados
If oMVCWS:PutXMLData()
 If oMVCWS:lPutXMLDataResult
  MsgInfo( 'Informação Importada com sucesso.' )
 Else
  MsgStop( 'Não importado' + CRLF + WSError() )
 EndIf
Else
 MsgStop( AllTrim( oMVCWS:cVldXMLDataResult ) + CRLF + WSError() )
EndIf
```

17.10 Web Services para modelos de datos que tienen dos ~~ó~~ o más entidades

Para la construcción de *Web Services* que tienen dos ~~ó~~ o más entidades lo que será diferente en el XML es que tendrá más niveles. Observe el fuente, ejemplo:

```
#INCLUDE 'PROTHEUS.CH'
#INCLUDE 'XMLXFUN.CH'
#INCLUDE 'FWMVCDEF.CH'

//-------------------------------------------------------------------
/*/{Protheus.doc} COMPW021
Ejemplo para utilizar un WebService generico para rutinas en MVC para una estructura de padre/hijo
@author Ernani Forastieri e Rodrigo Antonio Godinho @since 05/10/2009
@version P10
/*/
//-------------------------------------------------------------------

User Function COMPW021()
Local oMVCWS
Local cXMLEstrut := ''
```

Formatado: Português (Brasil)

Formatado: Espanhol (México)

Formatado: Espanhol (México)

Formatado: Espanhol (México)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
Local cXMLEsquema := ''
Local cXMLFile := '\XML\WSMVCTST.XML'
RpcSetType( 3 )
RpcSetEnv( '99', '01' )
// Instancia o WebService Generico para Rotinas em MVC
oMVCWS := WsFwWsModel():New()
oMVCWS:_URL := "http://127.0.0.1:8080/ws/FWWSMODEL.apw"
oMVCWS:cUserLogin := 'admin'
oMVCWS:cUserToken := 'admin'
oMVCWS:cPassword := ''
oMVCWS:cModelId := 'COMP021_MVC' // Fonte de onde se usara o Model
// Obtem a estutura dos dados do Model
If oMVCWS:GetXMLData()
If oMVCWS:GetSchema()
cXMLEsquema := oMVCWS:cGetSchemaResult
EndIf
  cXMLEstrut := oMVCWS:cGetXMLDataResult
  //<?xml version="1.0" encoding="UTF-8"?>
  //<COMP021MODEL Operation="1" version="1.01">
  //<ZA1MASTER modeltype="FIELDS" >
  //<ZA1_FILIAL order="1"><value></value></ZA1_FILIAL>
  //<ZA1_MUSICA order="2"><value></value></ZA1_MUSICA>
  //<ZA1_TITULO order="3"><value></value></ZA1_TITULO>
  //<ZA1_DATA order="4"><value></value></ZA1_DATA>
  // <ZA2DETAIL modeltype="GRID" >
  // <struct>
  // <ZA2_FILIAL order="1"></ZA2_FILIAL>
  // <ZA2_MUSICA order="2"></ZA2_MUSICA>
  // <ZA2_ITEM order="3"></ZA2_ITEM>
  // <ZA2_AUTOR order="4"></ZA2_AUTOR>
  // </struct>
  // <items>
  // <item id="1" deleted="0" >
  // <ZA2_FILIAL></ZA2_FILIAL>
  // <ZA2_MUSICA></ZA2_MUSICA>
  // <ZA2_ITEM></ZA2_ITEM>
  // <ZA2_AUTOR></ZA2_AUTOR>
  // </item>
  // </items>
  // </ZA2DETAIL>
  //</ZA1MASTER>
  //</COMP021MODEL>
  // Obtem o esquema de dados XML (XSD)
  If oMVCWS:GetSchema()
  cXMLEsquema := oMVCWS:cGetSchemaResult
  EndIf
  cXML := ''
  cXML += '<?xml version="1.0" encoding="UTF-8"?>'
  cXML += '<COMP021MODEL Operation="1" version="1.01">'
  cXML += '<ZA1MASTER modeltype="FIELDS">'
  cXML += '<ZA1_FILIAL order="1"><value>01</value></ZA1_FILIAL>'
  cXML += '<ZA1_MUSICA order="2"><value>000001</value></ZA1_MUSICA>'
  cXML += '<ZA1_TITULO order="3"><value>AQUARELA</value></ZA1_TITULO>'
  cXML += '<ZA1_DATA order="4"><value></value></ZA1_DATA>'
  cXML += ' <ZA2DETAIL modeltype="GRID" >'
  cXML += ' <struct>'
  cXML += ' <ZA2_FILIAL order="1"></ZA2_FILIAL>'
  cXML += ' <ZA2_MUSICA order="2"></ZA2_MUSICA>'
  cXML += ' <ZA2_ITEM order="3"></ZA2_ITEM>'
  cXML += ' <ZA2_AUTOR order="4"></ZA2_AUTOR>'
  cXML += ' </struct>'
  cXML += ' <items>'
  cXML += ' <item id="1" deleted="0" >'
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

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

```
   cXML += ' <ZA2_FILIAL>01</ZA2_FILIAL>'
   cXML += ' <ZA2_MUSICA>000001</ZA2_MUSICA>'
   cXML += ' <ZA2_ITEM>01</ZA2_ITEM>'
   cXML += ' <ZA2_AUTOR>000001</ZA2_AUTOR>'
   cXML += ' </item>'
   cXML += ' <item id="2" deleted="0" >'
   cXML += ' <ZA2_FILIAL>01</ZA2_FILIAL>'
   cXML += ' <ZA2_MUSICA>000002</ZA2_MUSICA>'
   cXML += ' <ZA2_ITEM>02</ZA2_ITEM>'
   cXML += ' <ZA2_AUTOR>000002</ZA2_AUTOR>'
   cXML += ' </item>'
   cXML += ' </items>'
   cXML += '</ZA2DETAIL>'
   cXML += '</ZA1MASTER>'
   cXML += '</COMP021MODEL>'

   // Joga o XML para o atributo do WebService
   oMVCWS:cModelXML := cXML

// Valida e Grava os dados
    If oMVCWS:PutXMLData()

       If oMVCWS:lPutXMLDataResult


            MsgInfo( 'Informação importada com sucesso.' )

       Else

            MsgStop( 'Não importado' + CRLF + WSError() )
       EndIf

    Else
         MsgStop( AllTrim( oMVCWS:cVldXMLDataResult ) + CRLF + WSError() )
    EndIf

Else

MsgStop( 'Problemas em obter Folha de Dados do Model' + CRLF + WSError() )

EndIf

RpcClearEnv()


Return NIL
//-------------------------------------
Static Function WSError()
Return IIf( Empty( GetWscError(3) ), GetWscError(1), GetWscError(3) )
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

18.Uso del comando New Model

Para facilitar el desarrollo fueron creados comandos que pueden generar, más simple y rápido una aplicación en MVC. El comando ***New Model.***

Este comando es el indicado para aquellas aplicaciones donde se tiene el uso de una tabla (antiguo ***Modelo1***) ~~ó~~ o una tabla no normalizada (encabezado e ítem del mismo registro), cuando se tenga la necesidad de trabajar con la estructura master-detail (antiguo ***Modelo2***) ~~ó~~ o donde se tenga el uso de dos tablas en una estructura ***master-detail*** (antiguo ***Modelo3***).

Utilizando el comando ***New Model*** no es necesario escribir todas las funciones y clases normalmente utilizadas en una rutina *MVC*. Durante el proceso de pre-compilación el ***New Model*** -utiliza directivas que son -transformados -en -un -fuente *MVC* -que -utiliza—FWmBrowse, ***ModelDef***, ***ViewDef*** y eventualmente ***MenuDef***.

Lo que se necesita para -utilizar este comando es que se tenga ~~una~~ uno de los modelos mencionados y que las estructuras de las tablas estén definidas en el diccionario SX3. No podrán ser construidas estructuras manualmente ~~ó~~ o agregar~~,~~ ~~ó quitar~~ retirar campos de las estructuras.

Como este comando es una directiva de compilación del tipo ***#COMMAND***, para utilizar este comando es necesario incluir la siguiente directiva en el fuente:

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

En el siguiente punto se definirá la sintaxis del comando y vendrán ejemplos de uso.

18.1 Sintaxis del New Model

Esta es la sintaxis del comando New Model:

NEW MODEL

TYPE <nType> ;

DESCRIPTION <cDescription> ;

BROWSE <oBrowse> ;

SOURCE <cSource> ;

MODELID <cModelID> ;

FILTER <cFilter> ;

CANACTIVE *<bSetVldActive>* ;

PRIMARYKEY *<aPrimaryKey>* ;

MASTER *<cMasterAlias>* ;

HEADER *<aHeader,...>* ;

BEFORE *<bBeforeModel>* ;

AFTER *<bAfterModel>* ;

COMMIT *<bCommit>* ;

CANCEL *<bCancel>* ;

BEFOREFIELD *<bBeforeField>* ;

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

AFTERFIELD *<bAfterField>* ;

LOAD *<bFieldLoad>* ;

DETAIL *<cDetailAlias>* ;

BEFORELINE *<bBeforeLine>* ;

AFTERLINE *<bAfterLine>* ;

BEFOREGRID *<bBeforeGrid>* ;

AFTERGRID *<bAfterGrid>* ;

LOADGRID *<bGridLoad>* ;

RELATION *<aRelation>* ;

ORDERKEY *<cOrder>* ;

UNIQUELINE *<aUniqueLine>* ;

AUTOINCREMENT *<cFieldInc>* ;

OPTIONAL

Donde:

TYPE <nType>

Tipo Numérico - Obligatorio
Tipo de Estructura
1 = 1 Tabela
2 = 1 Tabela Master/Detail
3 = 2 Tabelas Master/Detail

DESCRIPTION <cDescription>
Tipo Carácter - Obligatorio

Descripción de la Rutina

BROWSE <oBrowse>

Tipo Objeto - Obligatorio

Objeto de Browse que será utilizado

SOURCE <cSource>

Tipo Caracter - Obligatorio
Nombre del Fuente

MODELID <cModelID>

Formatado: Português (Brasil)

Tipo Carácter – Obligatorio

identificador (ID) del Model

FILTER <cFilter>

Tipo Carácter - Opcional

Filtro para Browse

CANACTIVE<bSetVldActive>

Tipo Bloque - Opcional

Bloco para validación en la activación del Model. Recibe como parámetro el Model Ex. { |oModel| COMP011ACT(oModel) }

PRIMARYKEY <aPrimaryKey>

Tipo Array - Opcional

Array con las llaves primarias del Browse, si no es informado buscará X2_UNICO de la tabla.

MASTER <cMasterAlias>

Tipo Carácter - Obligatorio

Tabla Principal (Master)

HEADER <aHeader>

Tipo Array - Obligatorio para TYPE = 2

Array con los campos que serán considerados en el "Encabezado"

BEFORE <bBeforeModel>

Tipo Bloque - Opcional

Bloco de Pre-Validación del Model. Recibe como parámetro el Model. Ex. { |oModel| COMP011PRE(oModel) }

AFTER <bAfterModel>

Tipo Bloque - Opcional

Bloco posterior a la validación del Model Recibe como parámetro el Model. Ex. { |oModel| COMP011POS(oModel) }

COMMIT <bCommit>

Tipo Bloque - Opcional

Bloque de Commit de los datos del Model. Recibe como parámetro el Model.

Ejemplo.{ |oModel| COMP022CM(oModel) }

CANCEL <bCancel>

Tipo Bloque - Opcional

~~Bloco~~ Bloque ejecutado en el botón de ~~cancelar~~anular. Recibe como parámetro el Model. Ejemplo. { |oModel| COMP011CAN(oModel) }

BEFOREFIELD <bBeforeField>

Tipo ~~Bloco~~ Bloque - Opcional

~~Bloco~~ Bloque de ~~Pre-Validación~~prevalidación del FORMFIELD de la tabla Master. Recibe como parámetro el ModelField, identificador (ID) del local de ejecución el identificador (ID) del ~~Formulário~~ formulario

Ejemplo.
{ |oMdlF,cId ,cidForm| COMP023FPRE(oMdlF,cId ,cidForm) }

AFTERFIELD <bAfterField>

Tipo Bloque - Opcional

Bloque después de la Validación de FORMFIELD de la tabla Master. Recibe como parámetro o ModelField, identificador (ID) del local de ejecución del identificador (ID) del ~~Formulário~~ formulario

Ejemplo.{ |oMdlF,cId ,cidForm| COMP023FPOS(oMdlF,cId ,cidForm) }

LOAD <bFieldLoad>

Tipo Bloque - Opcional

Bloque de Carga de los datos de FORMFIELD de la tabla Master

DETAIL <cDetailAlias>

Tipo Caracter - Obligatorio para TYPE = 2 ou 3 Tabla Detalle

BEFORELINE <bBeforeLine>

Tipo Bloque - Opcional

Bloque de ~~Pre-Validación~~prevalidación de la línea del FORMGRID de la tabla Detalle. Recibe como
parámetro el ModelGrid, el número de la línea del FORMGRID, la acción y el campo del FORMGRID.

Ex.{ |oMdlG,nLine,cAcao,cCampo| COMP023LPRE(oMdlG, nLine, cAcao, cCampo) }

Utilizado ~~solo~~ sólo para el TYPE = 2 ~~ó~~ o 3

AFTERLINE <bAfterLine>

Tipo Bloque – Opcional

Bloque ~~de~~ después de la Validación de la línea del FORMGRID de la tabla Detalle. Recibe como parámetro el ModelGrid y el número de la línea del FORMGRID.
Ej. { |oModelGrid, nLine| COMP022LPOS(oModelGrid, nLine) }

Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

BEFOREGRID <bBeforeGrid>

Tipo Bloque - Opcional

Bloque de ~~Pre-Validación~~prevalidación del FORMGRID de la tabla Detalle. Recibe como parámetro el ModelGrid

Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

AFTERGRID <bAfterGrid>

Tipo Bloque - Opcional

Bloco de ~~Pre-Validación~~prevalidación del FORMGRID de la tabla ~~Detalle~~detalle. Recibe como parámetro el ModelGrid
Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

LOADGRID <bGridLoad>

Tipo Bloque - Opcional

Bloque de Carga de los datos del FORMGRID de la tabla Detalle
Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

RELATION <aRelation>

Tipo Array – Obligatorio para TYPE = 2 o 3
Array bidimensional para la relación de las tablas Master/Detail
Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

ORDERKEY <cOrder>

Tipo Carácter - Opcional

Ordenar el FORMGRID de la tabla Detalle
Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

UNIQUELINE <aUniqueLine>

Tipo Array - Opcional

Array con campos que ~~podrán ser duplicados~~se pueden duplicar en el FORMGRID de la tabla Detalle

Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

AUTOINCREMENT <cFieldInc>

Tipo Array - Opcional
Campos auto incrementales para el FORMGRID de la tabla Detalle
Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

OPTIONAL

Indica si el llenado de FORMGRID de la tabla Detalle será opcional
Utilizado ~~solo~~ sólo para el TYPE = 2 o 3

Ejemplo:

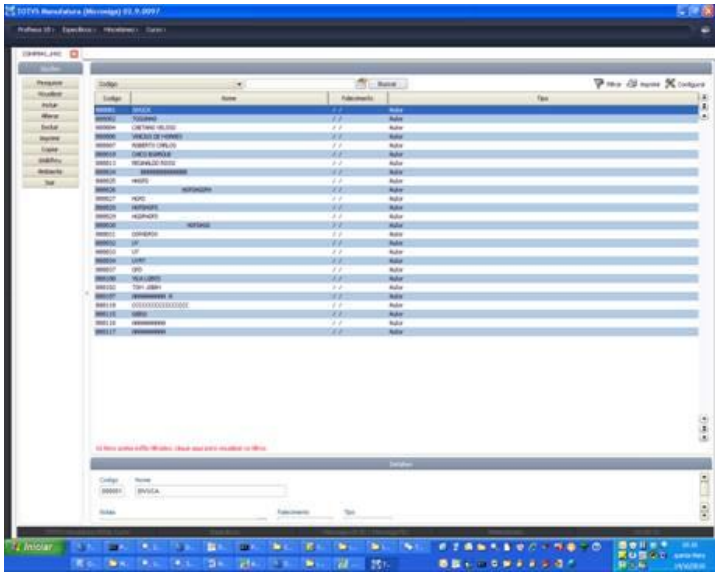
```
//
// Construcao para uma tabela
//
#INCLUDE "PROTHEUS.CH"
#INCLUDE "FWMVCDEF.CH"

User Function COMP041_MVC()
Local oBrowse

NEW MODEL ;
TYPE                    1 ;
DESCRIPTION "Cadastro de Autor/Interprete" ;
BROWSE                  oBrowse          ;
SOURCE                  "COMP041_MVC"    ;
MODELID                 "MDCOMP041"      ;
FILTER                  "ZA0_TIPO=='1'" ;
MASTER                  "ZA0"            ;
AFTER                   { |oMdl| COMP041POS( oMdl ) } ;
COMMIT                  { |oMdl| COMP041CMM( oMdl ) }
Return NIL


Static Function COMP041POS( oModel )
Help( ,, 'Help',, 'Acionou a COMP041POS', 1, 0 ) Return .T.

Static Function COMP041CMM( oModel )
FWFormCommit( oModel )
Return NIL
```

Formatado: Português (Brasil)

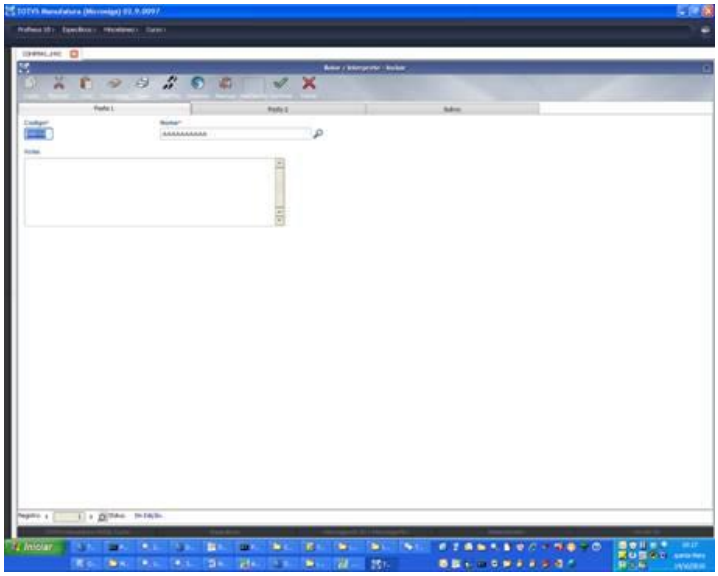
Formatado: Inglês (Estados Unidos)

Formatado: Inglês (Estados Unidos)

Visualmente tenemos:



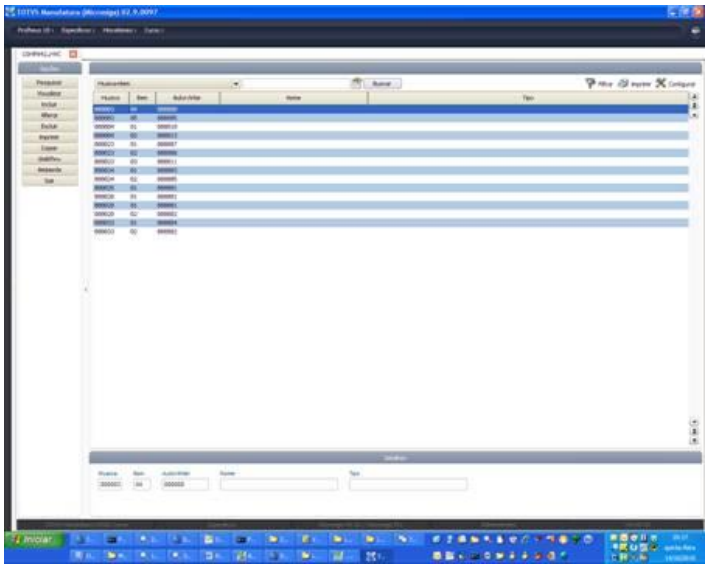
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®
Este manual é de propriedade da TOTVS. Todos os direitos reservados.®

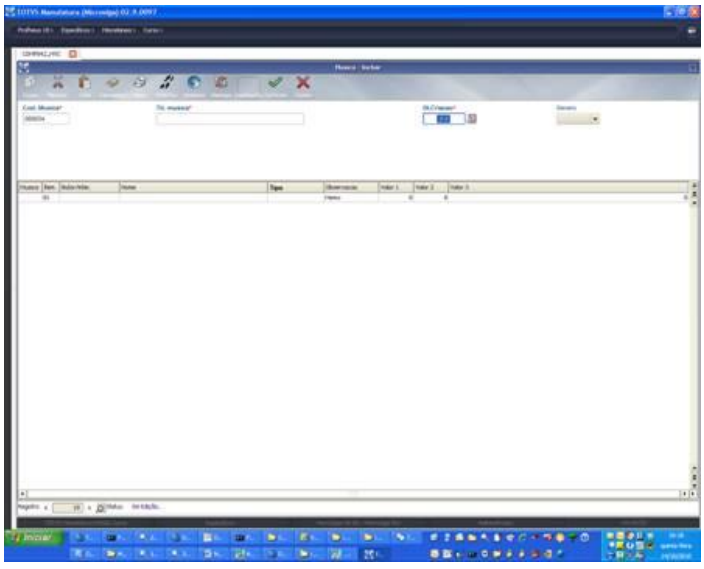


```
//
// Construcción para una tabla Master/Detail
//
#INCLUDE "PROTHEUS.CH"
#INCLUDE "FWMVCDEF.CH"

User Function COMP042_MVC()
Local oBrowse

NEW MODEL ;


TYPE2 ;

DESCRIPTION  "Tabela Nao Normalizada" ;

BROWSE     oBrowse;

SOURCE   "COMP042_MVC" ;

MODELID  "MDCOMP042" ;

MASTER   "ZA2";

HEADER{ 'ZA2_MUSICA', 'ZA2_ITEM' } ;

RELATION  { { 'ZA2_FILIAL', 'xFilial( "ZA2" )' }, ;

{ 'ZA2_MUSICA', 'ZA2_MUSICA' } } ;
```

```
UNIQUELINE{ 'ZA2_AUTOR' } ;
ORDERKEY ZA2->( IndexKey( 1 ) ) ;
AUTOINCREMENT  'ZA2_ITEM'
Return NIL
```

El Resultado es:

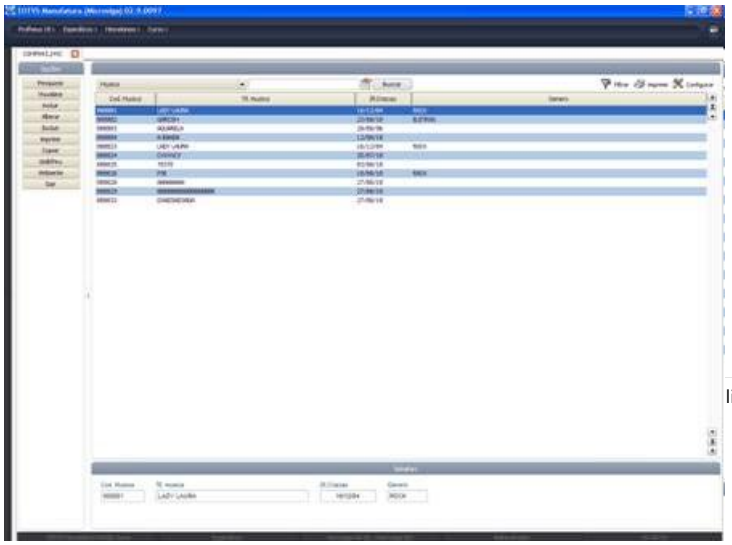




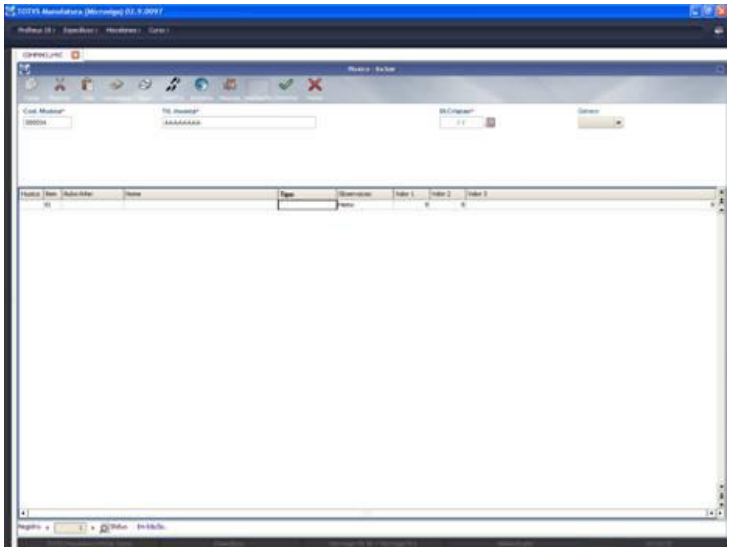
Formatado: Espanhol (Espanha - tradicional)

```
//
// Construcción para dos tablas Master/Detail
//
#INCLUDE "PROTHEUS.CH"
#INCLUDE "FWMVCDEF.CH"
User Function COMP043_MVC()

Local oBrowse

NEW MODEL ;

TYPE 3 ;

DESCRIPTION "Musicas";

BROWSE oBrowse ;

SOURCE "COMP043_MVC" ;

MODELID "MDCOMP043";

MASTER "ZA1";

DETAIL "ZA2";

RELATION { { 'ZA2_FILIAL', 'xFilial( "ZA2" )' }, ;
{ 'ZA2_MUSICA', 'ZA1_MUSICA' } } ;

UNIQUELINE{ 'ZA2_AUTOR' } ;

ORDERKEY ZA2->( IndexKey( 1 ) ) ;

AUTOINCREMENT      'ZA2_ITEM'
Return NIL
```

El Resultado es:



Formatado: Espanhol (Espanha - tradicional)



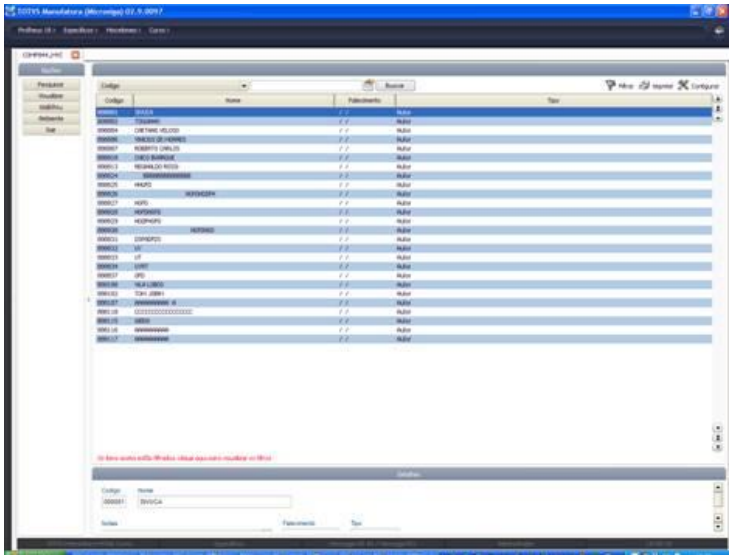
```
//
// Construcción para una tabla con menudef diferenciado
//
#INCLUDE "PROTHEUS.CH"
#INCLUDE "FWMVCDEF.CH"

User Function COMP044_MVC()

Local oBrowse

NEW MODEL ;

TYPE1 ;

DESCRIPTION "Cadastro de Autor/Interprete" ;

BROWSE   oBrowse;

SOURCE   "COMP044_MVC";

MENUDEF   "COMP044_MVC" ;
```

Formatado: Inglês (Estados Unidos)

```
MODELID   "MDCOMP044";

FILTER   "ZA0_TIPO=='1'" ;

MASTER   "ZA0"

Return NIL

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

Local aRotina := {}

ADD OPTION aRotina TITLE 'Pesquisar'   ACTION 'PesqBrw' OPERATION 1 ACCESS 0
ADD OPTION aRotina TITLE 'Visualizar' ACTION 'VIEWDEF.COMP044_MVC' OPERATION 2
ACCESS 0

Return aRotina
```

Formatado: Inglês (Estados Unidos)

El Resultado es:



19. Reutilizando un modelo de datos o interfaz ya existentes

Una de las grandes ventajas en la construcción de aplicaciones en *MVC* es la posibilidad de reutilización del modelo de dados (*Model*) o de la interfaz (*View*) en otras aplicaciones, utilizando el concepto de herencia.

~~Tanto podemos~~ Podemos reutilizar los componentes como están definidos, así como también podemos ~~adicionar~~ agregar ~~nuevas entidades~~nuevos entes a los mismos.

Para hacer esto, ~~necesitamos~~ dentro de la nueva aplicación es necesario instanciar el modelo de datos (*Model*) o de la interfaz (*View*).

A continuación veremos un ejemplo.

19.1 Reutilizando Componentes

En este ejemplo realizaremos el modelo de datos (*Model*) y la interfaz (View*)* ya existentes en una aplicación, para la construcción de una nueva aplicación sin modificaciones.

Utilizaremos las funciones:

FWLoadModel, ver cap. 0 12.4 Cargar el modelo de datos de una aplicación ya existente (FWLoadModel)) y~~;~~,

FWLoadView, ver cap. 0 12.5 Cargar la interfaz de una aplicación ya existente (*FWLoadView*).

En el ***ModelDef*** de la nueva aplicación instanciamos el modelo de datos (***Model***) de la aplicación ya existente:

```
Static Function ModelDef()
```

Local oModel := ***FWLoadModel***("COMP011_MVC")

```
Return oModel
```

En el ***MenuDef*** de la aplicación instanciamos la interface (View) de otra aplicación:

```
Static Function ModelDef()
Local oModel:= FWViewModel( "COMP011_MVC" ) Return oModel
```

En los ejemplos mostrados la nueva aplicación usará los mismos componentes de la aplicación que ya existe, en caso de que este definido en el ***ModelDef*** del fuente ***COMP011_MVC.***

Ejemplo:

```
#INCLUDE 'PROTHEUS.CH'
#INCLUDE 'FWMVCDEF.CH'
//-------------------------------------------------------------------
User Function COMP015_MVC()
Local oBrowse

oBrowse := FWMBrowse():New()
oBrowse:SetAlias('ZA0')
oBrowse:SetDescription('Cadastro de Autor/Interprete') oBrowse:DisableDetails()
```

Formatado: Português (Brasil)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)
Formatado: Inglês (Estados Unidos)

```
oBrowse:Activate()
Return NIL

//---------------------------------------------------------------------
Static Function MenuDef()
Return FWLoadMenuDef( "COMP011_MVC")
//---------------------------------------------------------------------
Static Function ModelDef()
// Creamos el modelo de dados de esta aplicación con un modelo existente en
// otra aplicación, en caso COMP011_MVC
Local oModel := FWLoadModel( "COMP011_MVC" ) Return oModel

//---------------------------------------------------------------------
Static Function ViewDef()
// Creamos el modelo de datos de esta aplicación con una interfaz existente en
// otra aplicación, en caso COMP011_MVC
Local oView := FWLoadView( "COMP011_MVC" )
Return oView
```

19.2 Reutilizando y complementando los componentes

Mostraremos ahora como reutilizar un componente de ***MVC*** donde ~~adicionamos nuevas entidades~~agregamos nuevos entes al mismo componente. ~~Solo~~ Sólo es posible ~~adicionar~~ agregar ~~nuevas entidades~~nuevos entes —y no es posible ~~quitar entidades~~eliminar entes , porque si ~~quitamos~~ retiramos ~~alguna entidad~~algún ente estaríamos rompiendo la regla de negocios construida en el modelo original.

Lo ideal para este tipo de uso es construir un modelo básico e incrementarlo conforme a las necesidades.

Analicemos primero el modelo de datos (*Model*). En el ejemplo a partir del modelo de datos ya existente agregaremos ~~una nueva entidad~~un nuevo ente.

El primer paso es crear la estructura ~~de la nueva entidad~~del nuevo ente, ver cap. 0 5.1 Construcción de una estructura de datos (***FWFormStruct***), para detalles.

```
// Crear la estructura que se agregará al Modelo de Dados
Local oStruZA6 := FWFormStruct( 1, 'ZA6', /*bAvalCampo*/, /*lViewUsado*/ )
```

Instanciamos el modelo que ya existe.

```
// Inicia el Model con un Model que ya existe
Local oModel              := FWLoadModel( 'COMP011_MVC' )
```

En nuestro ejemplo, agregamos un nuevo formulario, ver cap. 0 5.3 Creación de un Componente de formularios en el modelo de datos (AddFields), para detalles.

Note que en nuestra nueva aplicación no utilizamos el ***MPFormModel***, pues estamos ~~solo~~ sólo agregando ~~la entidad~~el ente. El ***MPFormModel*** fue utilizado en la aplicación original.

```
// Adiciona a nova FORMFIELD
oModel:AddFields( 'ZA6MASTER', 'ZA0MASTER', oStruZA6 )
```

Haremos la relación del nuevo formulario, ver cap. 0 6.5 Creación de la relación entre ~~las entidades~~los entes del modelo (SetRelation).

```
// Hace relacionamiento entre los componentes del model
oModel:SetRelation( 'ZA6MASTER', { { 'ZA6_FILIAL', 'xFilial( "ZA6"
)' }, { 'ZA6_CODIGO', 'ZA0_CODIGO' } }, ZA6->( IndexKey( 1 ) ) )
```

Asociamos la descripción del nuevo formulario.

```
// Adiciona a descriado del nuevo componente
oModel:GetModel( 'ZA6MASTER' ):SetDescription( 'Complemento dos Dados de
Autor/Interprete' )
```

Y al final retornamos el nuevo modelo.

```
Return oModel
```

Con esto creamos un modelo a partir de otro y agregamos un nuevo componente de formulario.

Veremos ahora como reutilizar la interfaz (*View*), también agregando un nuevo componente.

El primer paso es crear la estructura de la nueva entidad, ver cap. 0 5.1 Construcción de una estructura de datos (FWFormStruct).

```
// Crea la estructura a ser acrecentada  View
Local oStruZA6 := FWFormStruct( 2, 'ZA6' )
```

Instanciaremos el modelo utilizado por la interfaz, note que no instanciaremos el modelo original y si el modelo de la nueva aplicación que ya tiene el nuevo componente agregado en su modelo de datos.

```
// Crea un objeto de Modelo de Datos basado en el ModelDef del fuente informado
Local oModel := FWLoadModel( 'COMP015_MVC' )
```

Instanciamos la interfaz original

```
// Inicia la View con una View ya existente
Local oView := FWLoadView( 'COMP011_MVC' )
```

~~Adicionamos~~ Agregamos el nuevo componente de la view y asociamos la creación en el modelo, ver cap. 0 5.8 Creación de un componente de formularios en la interfaz (AddField), para más detalles.

```
// Adiciona en nuestra View un control de tipo FormFields(antigua enchoice)
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

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

```
oView:AddField( 'VIEW_ZA6', oStruZA6, 'ZA6MASTER' )
```

Tenemos que crear un ***box*** para el nuevo componente. Es necesario crear siempre un box vertical dentro de un horizontal y vice-versa como en ***COMP011_MVC*** el ***box*** que ya existe es horizontal, primero crearemos un vertical, para más detalles ver cap. 0 6.13 Mostrar los dados en la interfaz (CreateHorizontalBox / CreateVerticalBox).

```
// 'TELANOVA' es el box existente en la interfaz original
oView:CreateVerticallBox( 'TELANOVA' , 100, 'TELA'                )

// Nuevos Boxes
oView:CreateHorizontalBox( 'SUPERIOR' , 50, 'TELANOVA' )
oView:CreateHorizontalBox( 'INFERIOR' , 50, 'TELANOVA' )
```

Relacionar los componentes con lo box, ver cap. 0 5.10 Relacionando el componente de la interfaz (SetOwnerView).

```
oView:SetOwnerView( 'VIEW_ZA0', 'SUPERIOR' )
oView:SetOwnerView( 'VIEW_ZA6', 'INFERIOR' )
```

Y al final retornamos el nuevo objeto de la interfaz.

```
            Return oView
```

Con esto creamos una *interfaz* a partir de otra y agregamos un nuevo componente.

Un ejemplo de aplicación para este concepto seria la internacionalización, donde podríamos tener un modelo básico e incrementaríamos conforme a la localización.

Para entender mejor la internacionalización, vea el Apéndice A.

A continuación, tenemos el ejemplo completo de la aplicación que reutiliza componentes.

19.3 Ejemplo completo de una aplicación que reutiliza componentes del modelo e interfaz

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

//-------------------------------------------------------------------
User Function COMP015_MVC()
Local oBrowse

oBrowse := FWMBrowse():New()
oBrowse:SetAlias('ZA0')
oBrowse:SetDescription( 'Cadastro de Autor/Interprete' )
oBrowse:AddLegend( "ZA0_TIPO=='1'", "YELLOW", "Autor")
oBrowse:AddLegend( "ZA0_TIPO=='2'", "BLUE" , "Interprete")
oBrowse:Activate()
```

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
Return NIL

//-------------------------------------------------------------------
Static Function MenuDef()
Local aRotina := {}
ADD OPTION aRotina TITLE 'Visualizar' ACTION 'VIEWDEF.COMP015_MVC' OPERATION 2
ACCESS 0
ADD OPTION aRotina TITLE 'Incluir'ACTION   'VIEWDEF.COMP015_MVC'   OPERATION   3
ACCESS 0
ADD OPTION aRotina TITLE 'Alterar'ACTION   'VIEWDEF.COMP015_MVC'   OPERATION   4
ACCESS 0
ADD OPTION aRotina TITLE 'Excluir'ACTION   'VIEWDEF.COMP015_MVC'   OPERATION   5
ACCESS 0
ADD OPTION aRotina TITLE 'Imprimir'     ACTION 'VIEWDEF.COMP015_MVC' OPERATION 8
ACCESS 0
ADD OPTION aRotina TITLE 'Copiar' ACTION   'VIEWDEF.COMP015_MVC'   OPERATION   9
ACCESS 0
Return aRotina

//-------------------------------------------------------------------
Static Function ModelDef()
// Crea la estructura a ser acrecentada en el Modelo de Datos
Local oStruZA6 := FWFormStruct( 1, 'ZA6', /*bAvalCampo*/,/*lViewUsado*/ )

// Inicia el Model con un Model ya existente
Local oModel := FWLoadModel( 'COMP011_MVC' )

// Adiciona una nueva FORMFIELD
oModel:AddFields( 'ZA6MASTER', 'ZA0MASTER', oStruZA6 )
// Hace relacionamiento entre los componentes del model
oModel:SetRelation( 'ZA6MASTER', { { 'ZA6_FILIAL', 'xFilial( "ZA6" )' }, {
'ZA6_CODIGO', 'ZA0_CODIGO' } }, ZA6->( IndexKey( 1 ) ) )
// Adiciona la descripción del nuevo componente
oModel:GetModel( 'ZA6MASTER' ):SetDescription( 'Complemento dos Dados de
Autor/Interprete' )


Return oModel

//-------------------------------------------------------------------
Static Function ViewDef()
// Crea un objeto de Modelo de Dados basado en ModelDef en el fuente informado
Local oModel := FWLoadModel( 'COMP015_MVC' )
// Crea la estructura a ser acrecentada en la View
Local oStruZA6 := FWFormStruct( 2, 'ZA6' )
// Inicia a View com una View ya existente
Local oView := FWLoadView( 'COMP011_MVC' )

// Modifica el Modelo de datos que será utilizado
oView:SetModel( oModel )
// Adiciona en nuestra View un control de tipo FormFields(antigua enchoice)
oView:AddField( 'VIEW_ZA6', oStruZA6, 'ZA6MASTER' )

// Es necesario crear siempre un box vertical dentro de uno horizontal y vice-
```

Formatado: Espanhol (Espanha - tradicional)

Formatado: Espanhol (Espanha - tradicional)

Formatado: Espanhol (Espanha - tradicional)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

Formatado: Português (Brasil)

```
versa
// Como en la COMP011_MVC o box é horizontal, crea un vertical primero
// Box existente en la interfaz original
oView:CreateVerticallBox( 'TELANOVA' , 100, 'TELA'  )


// Nuevos Boxes
oView:CreateHorizontalBox( 'SUPERIOR' , 50, 'TELANOVA' )
oView:CreateHorizontalBox( 'INFERIOR' , 50, 'TELANOVA' )

// Relaciona el identificador (ID) da View com o "box" para exibicao
oView:SetOwnerView( 'VIEW_ZA0', 'SUPERIOR' )
oView:SetOwnerView( 'VIEW_ZA6', 'INFERIOR' )


Return oView
```


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

Apéndice A

El Framework *MVC* de Microsiga Protheus y la internacionalización.

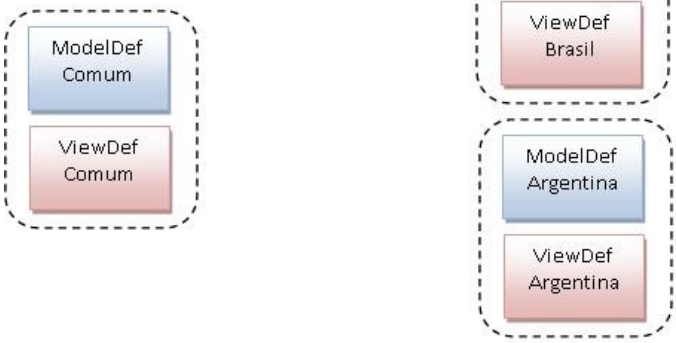
Internacionalización (I18N) y localización (L10N) son procesos de desarrollo y/o adaptación de software, para una lengua y/~~u~~ o cultura de un país. La internacionalización de un software no establece un nuevo Sistema, solamente adapta ~~las~~ los mensajes y etiquetas del Sistema a la lengua y cultura locales. La localización a su vez, agrega nuevos elementos del país al Sistema, como procesos, aspectos legales, entre otros.
El Framework MVC ~~auxilia~~ apoya a la localización del Sistema, formando ~~de~~ en componentes ~~al~~ el software, de tal forma que la parte ~~de un~~ común a todos los países ~~divide~~ se desagregue de la parte no común, incluyendo su interfaz y regla de negocio.

Por ejemplo, tome como base el formulario ***Nota Fiscal/Invoice.*** Este formulario tiene como característica común en todos los países los elementos: **Origen, Destino, Lista de productos, Transporte y Facturas.**

En ciertos países como en Brasil, es necesario registrar elementos legales, **como impuestos, escrituración, códigos de clasificación**, entre otros. La alternativa que se tiene es duplicar el código o ~~alterar~~ modificar el código insertando líneas de código de los elementos localizados. A pesar de esta alternativa, funciona bien ~~en el inicio~~a corto plazo, a ~~lo~~ largo ~~del tiempo~~plazo esto no es práctico debido al volumen de implementaciones diferentes para cada país, lo que ~~causando~~ causa grandes problemas y un alto costo para ~~la sustentación del~~mantener el Sistema.

El Framework *MVC* trae una luz racional y simple para este problema. La herencia de formularios. Es posible construir un formulario común para la ~~**Nota Fiscal**~~**Factura/Invoice** que no tenga ningún elemento de localización y utilizarlo por herencia, como base para los formularios localizados.

Este modelo, garantiza la evolución de la localización y de la parte común del formulario sin que una implementación afecte a otra reduciendo el costo de ~~sustentación~~ mantenimiento del producto.



La herencia del Framework *MVC* ~~pode~~ puede ocurrir en el Model y la View o solamente en View.

En este momento ~~usted~~ se debe estar preguntando ¿Cómo es posible esto?. La respuesta está en ~~una~~ un par de funciones ***FWLoadModel*** y ***FWLoadView***, como puede observar en el código a continuación:

```
#INCLUDE "MATA103BRA.CH"

Static Function ModelDef()
Local     oModel     :=     FWLoadModel("MATA103")
oModel:AddField(....)
oModel:AddGrid(....)
Return(oModel)

Static Function ViewDef()
Local     oView     :=     FWLoadView("MATA103")
oView:AddField(…)
oView:AddGrid(…)
Return (oView)
```

Entre las ~~innumeradas~~ innumerables ventajas que tiene este modelo de desarrollo nos gustaría destacar ~~-~~la creación de componentes, y el aislamiento ~~de~~ del código fuente. El aislamiento permite que los dos códigos fuente~~s~~ evolucionen por separado, sin embargo por ~~la herencia~~herencia el código localizado siempre ~~irá heredar~~heredará los beneficios de la parte común, incluso ~~posibilitara a~~permite que las dos personas interactúen simultáneamente en el mismo proceso sin que una perjudique el trabajo de la otra.

Glosario

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