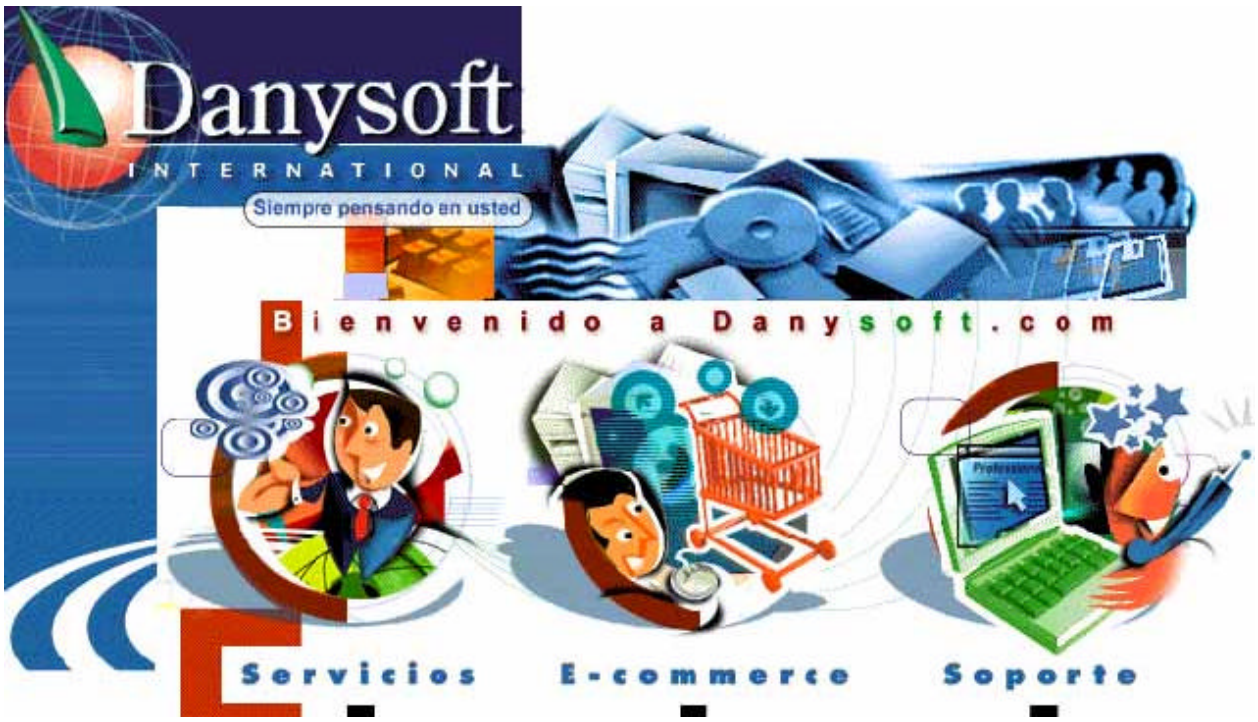
**Artículos técnicos Grupo Danysoft:**

# Reglas de negocio en aplicaciones Delphi




**Por Pablo Reyes – Equipo Grupo Danysoft**
**enero de 2002 - (902) 123146**
**www.danysoft.com**

Este documento se ha realizado utilizando *Doc-To-Help*®, distribuido por :


Danysoft Internacional
Avda de España 17
28100 Alcobendas – Madrid
Tfno. 902.123146
Fax. 902.123145
http://www.danysoft.com
http://www.danyshop.com
danysoft@danysoft.com

# Reglas de negocio en aplicaciones Delphi

Mas allá de lo que la teoría dice muchas veces nos vemos obligados a codificar reglas de negocio en nuestras aplicaciones. En este artículo veremos las facilidades que ofrece Delphi para la implementación de reglas de negocio explorando las propiedades y eventos de las clases TDataSet y TField y sus descendientes.

## *¿Qué son las reglas de negocio?*

En el libro **Database Design for Mere Mortals**, *Michael J. Hernandez* define las reglas de negocio con la siguiente frase:

> ***"Las reglas de negocio imponen restricciones o limitaciones sobre ciertos aspectos de una base de datos basadas en la manera en que la organización persive o utiliza sus datos."***

Si bien la teoría dice que las reglas de negocio no deben residir en una aplicación cliente es muy dificil desarrollar una aplicación para el mundo real que no contenga ni una sola regla de negocio. La principal motivación para hacerlo suele ser el rendimiento, aunque no hay que dejar de lado cuestiones como lograr una interfase de usuario amigable o restricciones al momento de modificar la base de datos.
Lo concreto es que, de una u otra forma, tarde o temprano, deberemos incluir reglas de negocio en nuestras aplicaciones por lo que es bueno que sepamos cual es la mejor forma de hacerlo. Asi que, manos a la obra.

## *Componentes de acceso a datos*

Delphi 6 ofrece varias alternativas a la hora de acceder a una base de datos. Podemos utilizar los componentes basados en el Borland Database Engine, los basados en el DBExpress o los basados en ADO. También podemos utilizar los componentes IBExpress para acceder directamente a InterBase e incluso componentes de terceros para acceder directamente a distintos motores o aprovechar funcionalidades adicionales.
Lo cierto es que todos estos componentes tienen un ancestro común que es la clase abstracta TDataSet.
La clase TDataSet provee la funcionalidad básica para administrar un conjunto de datos permitiendo, salvo algunas excepciones, recorrerlo en ambas direcciones insertando, modificando y borrando registros.
Los objetos descendientes de TDataSet poseen una colección de objetos del tipo TField cada uno de los cuales representa una columna del conjunto de datos. Esta colección podemos crearla explícitamente tanto en tiempo de diseño como en tiempo de ejecución o dejar que Delphi la cree por nosotros.
Idealmente los componentes de acceso a datos deberían residir en un DataModule y todas las reglas de negocio que los afecten deberían ser codificadas en el mismo DataModule. De esta forma podremos aislarlas de la lógica visual de nuestra aplicación, es decir, de la interfase de usuario.
Delphi provee componentes para visualizar datos de los componentes de acceso a datos y muchos programadores, sobre todo los principiantes, cometen el error de codificar reglas de negocio en los eventos de dichos componentes mezclándolas y atándolas a la lógica visual. Por ejemplo, es muy común ver como validaciones a nivel de campo son codificadas en sus correspondientes componentes DBEdit en el evento OnExit.
A grandes rasgos podemos dividir las reglas de negocio, de acuerdo a su implementación en Delphi, en dos grandes grupos: reglas de negocio a nivel de registro y reglas de negocio a nivel de campo. La mayoría de las reglas de negocio a nivel de

registro son implementadas en los componentes DataSet mientras que la mayoría de las reglas de negocio a nivel de campo son implementadas en los objetos TField.

## *De los eventos*

Es importante hacer un par de aclaraciones previas relacionadas con los eventos. Los eventos, tanto de un DataSet como de un TField, pueden ser generados por acción directa o indirecta tanto del usuario como de la aplicación. Por ejemplo:

- **por acción directa del usuario:** si el registro actual está siendo modificado y el usuario hace clic en el botón Post del componente DBNavigator correspondiente.
- **por acción indirecta del usuario:** si el registro actual está siendo modificado y el usuario hace clic en el botón Next del componente DBNavigator correspondiente
- **por acción directa de la aplicación:** si el registro actual está siendo modificado y por código llamamos al método Post del DataSet correspondiente.
- **por acción indirecta de la aplicación:** si el registro actual está siendo modificado y por código llamamos al método First del DataSet correspondiente.

En cualquier caso, lo importante es saber que los eventos pueden ser generados por diversas acciones y que, generalmente, estas acciones generan una serie de eventos en cadena y que la única manera de detener esta cadena es generar una excepción. Por ejemplo, al llamar al método Post de un DataSet se generan los eventos BeforePost y AfterPost en ese orden. Si codificamos una validación en el evento BeforePost y la misma es violada, la única forma de detener la cadena de eventos, es decir, que el evento AfterPost sea generado y la grabación del registro sea llevada a cabo, es generando una excepción.
Es importante tener mucho cuidado con lo que codificamos en los eventos ya que es muy fácil cometer errores y hacer que la aplicación entre en un bucle infinito. Por ejemplo, el siguiente código hace que la aplicación entre en un bucle infinito:

```
procedure Form1.Table1BeforePost(DataSet: TDataSet);
begin
  DataSet.First;
end;
```

En el ejemplo anterior, la llamada al método First del DataSet que generó el evento BeforePost provoca una nueva llamada al método Post del mismo DataSet (realizada implícitamente por el método First) que generará nuevamente el evento BeforePost y asi la aplicación entrará en un bucle infinito.

## *DataSet: Reglas de negocio a nivel de registro*

Las reglas de negocio a nivel de registro son aquellas que deben ser aplicadas antes de grabar los cambios realizados a un registro, ya sea por modificar un registro existente o por insertar uno nuevo. Las principales propiedades y eventos son las siguientes.

### La propiedad Constraints

Los componentes basados en el BDE (Table, Query y BDEClientDataSet), los de acceso directo a InterBase (IBTable, IBQuery y IBClientDataSet) y el componente ClientDataSet poseen la propiedad constraints que mantiene una colección de validaciones a nivel de registro. Las reglas de negocio son procesadas en el orden en que están en la colección.

Las principales propiedades de cada validación son CustomConstraint y ErrorMessage. La propiedad CustomConstraint permite indicar una validación a nivel de registro utilizando sintaxis SQL. Por ejemplo, las siguientes reglas de negocio son todas válidas:

```
Nombre IS NOT NULL AND Edad > 21
Nombre LIKE 'A%'
Edad BETWEEN 30 AND 40
```

La propiedad ErrorMessage contiene el mensaje de error correspondiente. Cuando una validación es violada Delphi genera una excepción con dicho mensaje.
Al activar el DataSet Delphi realiza un pre-procesamiento de las reglas de negocio para verificar que sean correctas, es decir, que su sintaxis sea correcta, que hagan referencia a campos existentes y demás. Si alguna regla de negocio no es correcta entonces Delphi genera una excepción.

## Los eventos BeforeInsert, BeforeEdit y BeforeDelete

Estos eventos están presentes en casi todos los descendientes de TDataSet, salvo los basados en DBExpress que, por representar conjuntos de datos de sólo lectura, no los poseen a excepción, claro está, del componente SQLClientDataSet.

La sintaxis es la siguiente:

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
property BeforeInsert: TDataSetNotifyEvent;
property BeforeEdit: TDataSetNotifyEvent;
property BeforeDelete: TDataSetNotifyEvent;
```

El parámetro DataSet es el DataSet que generó el evento.

Estos eventos se generan, como su nombre lo indica, antes de insertar, editar y borrar el registro actual. La mayoría de los descendientes de TDataSet poseen una propiedad llamada State que indica el estado del DataSet. Algunos de estos estados son dsBrowse, dsInsert y dsEdit. Los eventos BeforeInsert y BeforeEdit se generan antes de que el estado del DataSet sea modificado, es decir, de dsBrowse a dsInsert o dsEdit según corresponda.
Las validaciones más comunes codificadas en estos eventos son las relacionadas con permisos de usuario, es decir, validar si el usuario actual tiene permisos para insertar, editar o borrar registros y, en el caso de que no los tenga, cancelar la acción mediante una excepción.

## El evento OnNewRecord

Este evento está presente en casi todos los descendientes de TDataSet, salvo los basados en DBExpress que, por representar conjuntos de datos de sólo lectura, no lo poseen a excepción, claro está, del componente SQLClientDataSet.

La sintaxis es la siguiente:

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
property OnNewRecord: TDataSetNotifyEvent;
```

El parámetro DataSet es el DataSet que generó el evento.

Quizás muchos piensen que el evento OnNewRecord no tiene nada que ver con reglas de negocio pero yo creo que si.
Este evento se genera cada vez que se inserta o agrega un registro nuevo. Tiene la particularidad de que los valores asignados a los campos de un DataSet en su evento OnNewRecord no son considerados modificaciones al registro. La mayoría de los descendientes de TDataSet poseen una propiedad llamada Modified que indica si el registro actual fue modificado. Si asignamos valores por defecto a los campos de un DataSet en su evento OnNewRecord el valor de la propiedad Modified sigue siendo False.
En este evento no se suelen codificar validaciones sino, como se dijo en el párrafo anterior, valores por defecto para los campos que asi lo necesiten.

## El evento BeforePost

Este evento está presente en casi todos los descendientes de TDataSet, salvo los basados en DBExpress que, por representar conjuntos de datos de sólo lectura, no lo poseen a excepción, claro está, del componente SQLClientDataSet.

La sintaxis es la siguiente:

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
property BeforePost: TDataSetNotifyEvent;
```

El parámetro DataSet es el DataSet que generó el evento.

El evento BeforePost se genera antes de que los cambios al registro actual, correspondientes a una modificación o inserción, sean grabados.
Las validaciones más comunes codificadas en este evento son las que tienen que ver con validaciones a nivel de registro. En cierto sentido el evento BeforePost es similar a la propiedad Constraints aunque mucho más potente ya que no estamos limitados a sentencias SQL. El evento BeforePost es el más utilizado de todos.

## Los eventos AfterPost y AfterDelete

Estos eventos están presentes en casi todos los descendientes de TDataSet, salvo los basados en DBExpress que, por representar conjuntos de datos de sólo lectura, no lo poseen a excepción, claro está, del componente SQLClientDataSet.

La sintaxis es la siguiente:

```
type TDataSetNotifyEvent = procedure(DataSet: TDataSet) of object;
property AfterPost: TDataSetNotifyEvent;
property AfterDelete: TDataSetNotifyEvent;
```

El parámetro DataSet es el DataSet que generó el evento.

Como su nombre lo indica, estos eventos se generan inmediatamente después de que un registro fue grabado o borrado según corresponda.
Las reglas de negocio más comunes que se codifican en estos eventos son las relacionadas con pistas de auditoria. Por ejemplo, cada vez que se borra un registro de la

tabla Clientes debemos dar de alta un registro en la tabla de auditoria indicanado la clave primaria del registro borrado, la fecha y hora y el usuario.
Estos eventos también son muy utilizados en relaciones maestro/detalle cuando se realizan modificaciones en las tablas detalle. Por ejemplo, al agregar un ítem al detalle de una factura necesitamos actualizar los totales de la misma.

## *TField: Reglas de negocio a nivel de campos*

Los componentes descendientes de TDataSet poseen una colección de objetos del tipo TField cada uno de los cuales representa una columna del conjunto de datos y poseen propiedades y eventos que permiten codificar reglas de negocio a nivel de campos.

### Las propiedades CustomConstraint y ConstraintErrorMessage

La propiedad CustomConstraint permite indicar una validación a nivel de campo utilizando sintaxis SQL. Por ejemplo:

```
Nombre IS NOT NULL AND Edad > 21
```

La propiedad ConstraintErrorMessage contiene el mensaje de error correspondiente. Cuando la validación sea violada Delphi generará una excepción con dicho mensaje.
Al activar el DataSet Delphi realiza un pre-procesamiento de las reglas de negocio para verificar que sean correctas, es decir, que su sintaxis sea correcta y demás. Si alguna regla de negocio no es correcta entonces Delphi genera una excepción.

### La propiedad DefaultExpression

Esta propiedad permite indicar un valor por defecto para el campo utilizando sintaxis SQL. Por ejemplo:

```
-1
```

Utilizar la propiedad DefaultExpression es similar a asignarle un valor al campo en el evento del DataSet OnNewRecord pero con muchas más limitaciones.

### La propiedad ReadOnly

Esta propiedad permite indicar si el valor del campo puede ser modificado. Los componentes visuales de acceso a datos hacen honor a esta propiedad permitiendo que el valor del campo sea modificado o no según corresponda.
También el código fuente hace honor a esta propiedad generando una excepción si intentamos modificar el valor de un campo que tiene su propiedad ReadOnly con el valor True.

### La propiedad Required

Esta propiedad indica si el valor del campo es requerido, es decir, si el valor del campo acepta valores nulos. Esta es una validación a nivel de campo pero que se aplica cuando se graba el registro. El valor nulo depende del tipo de dato del campo. Por ejemplo, en el caso de un campo que representa un valor entero, el valor nulo correspondiente es 0.

## Propiedades de campos numéricos

Los campos con tipos de datos numéricos, concretamente los descendientes de TNumericField en sus distintas variantes, poseen las siguientes propiedades:

- **DisplayFormat:** permite indicar el formato con el cual el valor del campo debe ser mostrado por los componentes visuales.
- **EditFormat:** permite indicar el formato con el cual el valor del campo debe ser editado en los componentes visuales.
- **MaxValue:** permite indicar el valor máximo que el campo acepta.
- **MinValue:** permite indicar el valor mínimo que el campo acepta.

## El evento OnValidate

La sintaxis es la siguiente:

```
type TFieldNotifyEvent = procedure(Sender: TField) of object;
property OnValidate: TFieldNotifyEvent;
```

El parámetro Sender es el Field que generó el evento.

Los valores de los campos son almacenados temporariamente en memoria antes de ser aplicados en la base de datos. Este evento se genera inmediatamente antes de que el valor del campo sea escrito en su almacenamiento temporario.
El evento OnValidate es el lugar por excelencia para codificar validaciones a nivel de campo. Sin embargo, es importante tener en cuenta la navegabilidad de la interfase de usuario ya que validaciones muy estrictas podrían convertir a una interfase amigable en una que los usuarios rechacen. Por ejemplo, si codificamos en el evento OnValidate una validación para que no acepte valores nulos y el usuario se posiciona en el componente visual del campo entonces no podrá avandanarlo hasta que ingrese un valor.

## El evento OnChange

La sintaxis es la siguiente:

```
type TFieldNotifyEvent = procedure(Sender: TField) of object;
property OnChange: TFieldNotifyEvent;
```

El parámetro Sender es el Field que generó el evento.
Los valores de los campos son almacenados temporariamente en memoria antes de ser aplicados en la base de datos. Este evento se genera inmediatamente después de que el valor del campo es escrito en su almacenamiento temporario.
Las validaciones más comunes codificadas en este evento son aquellas acciones que deben ser llevadas a cabo cuando el valor de un campo es modificado. Por ejemplo, si el valor de un campo depende del valor de otro campo entonces en el evento OnChange del segundo campo debe codificarse el cálculo del valor del primero.

## Problemas de idioma

Lamentablemente no existe una versión de Delphi en español, aunque si existiera no creo que solucionaría totalmente los problemas de idioma.
Muchas reglas de negocio pueden ser controladas por medio de propiedades con la ventaja de que reducen el código a escribir. Sin embargo, cuando una de estas reglas es

violada Delphi se encarga de generar una excepción cuyo mensaje de error está en inglés.
Una solución a este problema puede ser navegar la VCL y modificar el código fuente para que los mensajes de error aparezcan en nuestro idioma. Si bien no es una tarea muy compleja, es muy laboriosa y seguramente tendremos que repetirla para cada nueva versión de Delphi.
Otra solución es capturar las excepciones en un lugar centralizado y generar una excepción equivalente pero con el mensaje error en nuestro idioma. Tampoco es una tarea muy compleja pero si muy laboriosa y no siempre viable. Una ventaja frente a la solución anterior es que no tendremos que repetirla para cada nueva versión de Delphi.
Por último, la solución más elegante a mi criterio es no implementar reglas de negocio por medio de propiedades sino codificarlas en los eventos correspondientes y generar excepciones con mensajes de error en nuestro idioma. Esto implica más código a escribir pero también implica más control sobre el comportamiento de nuestra aplicación.

## *Lo que no debemos hacer*

A continuación una serie de consejos sobre lo que no debemos hacer:

- Aunque la tentación es grande y el sentido común puede engañarnos debemos evitar codificar reglas de negocio en componentes visuales. Por ejemplo, si queremos dar de alta un registro en una tabla de auditoria cada vez que se borre un registro de la tabla Clientes no lo hagamos en el evento BeforeAction del componente DBNavigator asociado. El mejor lugar para hacerlo es el evento AfterDelete del DataSet.
- Coloquemos todos los componentes de acceso a datos en un DataModule y codifiquemos las reglas de negocio en el mismo DataModule. Hagamos que los formularios visuales sepan acerca del DataModule pero que el DataModule no sepa nada acerca de los formularios visuales. De esta manera no condicionaremos las reglas de negocio a la interfase de usuario sino todo lo contrario.
- En lo posible tratemos de usar las propiedades y eventos genericos de la clase TDataSet. Esto hará más fácil el reemplazo de componentes de acceso a datos si nos vemos obligados de migrar de una tecnología de acceso a datos a otra, por ejemplo, de el BDE a ADO.

## *Conclusiones*

Hemos visto de que manera las reglas de negocio pueden ser codificadas en una aplicación Delphi por medio de los eventos y las propiedades de las clases TDataSet y TField y sus descendientes.
Las reglas de negocio deben ser codificadas en forma independiente de la interfase visual de la aplicación y no deben ser condicionadas por ellas sino todo lo contrario, las reglas de negocio deben condicionar la interfase visual. El lugar ideal para codificar las reglas de negocio son los mismos DataModule en donde los componentes de acceso a datos residen.