E-Sniper Software Development Kit

Table of contents

Data model

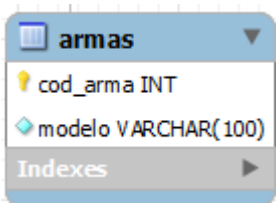
The E-Sniper tables are arranged in a star model, with the shots as the fact table and the shooting parameters and dates as the dimension tables.

Shooting parameters

These tables contain the shooting parameters.

Gun model table

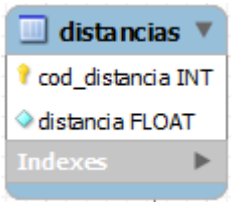
The table **armas** contains the different models of guns, the fields are as follows:



- **Cod_arma:** the primary key, an integer with auto increment.
- **Modelo:** the name of the gun model.

Distances table

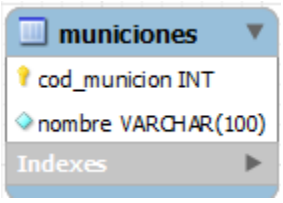
The table **distancias** contains the distances from the shooter to target, the fields are as follows:



- **cod_distancia:** the primary key, an integer with auto increment.
- **Distancia**: the distance to the target, a float value, normally in meters or yards.

Ammunition type table

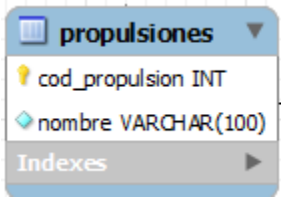
The table **municiones** contains the different types of ammo used in shooting, the fields are as follows:



- **cod_municion:** the primary key, an integer with auto increment.
- **Nombre:** the name of the type of ammunition.

Propellant type table

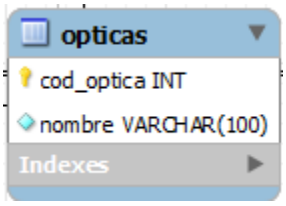
The table **propulsiones** contains the different types of propellants used by the guns, the fields are as follows:



- **cod_propulsion:** the primary key, an integer with auto increment.
- **Nombre:** a descriptive name for the propellant type.

Optics table

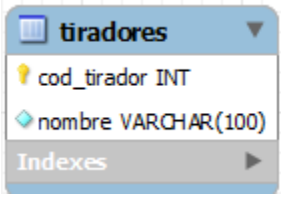
The table **opticas** contains the different types of optics used with the guns, the fields are as follows:



- **cod_opetica:** the primary key, an integer with auto increment.
- **Nombre:** a descriptive name for the type of optics.

Shooters table

The table **tiradores** contains the names of the shooters, the fields are as follows:



- **cod_tirador:** the primary key, an integer with auto increment.
- **Nombre:** the name of the shooter.

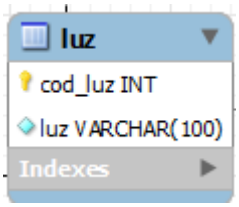
Trigger weights table

The table **presion_gatillo** contains values for the different trigger weights, the fields are as follows:

- **cod_presion:** the primary key, an integer with auto increment.
- **Presion:** a float value for the trigger weight, regardless of the measurement units.
- **Na:** this value can be 0 or 1, there are only a register with a value of 1 in this field, and indicates that this parameter has no value. The user only can create registers with this field with a value of 0, the na register is provided for the system by default.

Ambient light table

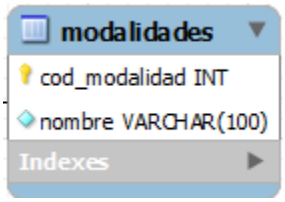
The table **luz** contains the different types of ambient light when shooting, the fields are as follows:



- **cod_luz:** the primary key, an integer with auto increment.
- **Luz:** a descriptive name for the ambient light.

Shot classification table

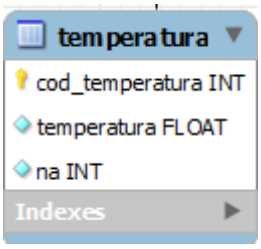
The table **modalidades** contains extra parameters to classify the shots in a free way, the fields are as follows:



- **cod_modalidad:** the primary key, an integer with auto increment.
- **Nombre:** a descriptive name for the class.

Temperature table

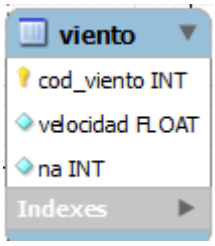
The table **temperatura** contains different values for the temperature; this is useful for GBB guns using green gas as propellant. The fields are as follows:



- **cod_temperatura:** the primary key, an integer with auto increment.
- **Temperatura:** a float value with the value of temperature, no matter the measurement units.
- **Na:** this value can be 0 or 1, there are only a register with a value of 1 in this field, and indicates that this parameter has no value. The user only can create registers with this field with a value of 0, the na register is provided for the system by default.

Wind speed table

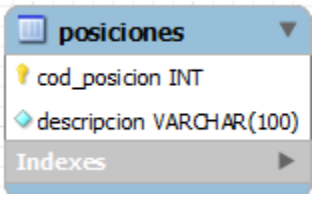
The table **viento** contains different values for the wind speed, the fields are as follows:



- **cod_viento:** the primary key, an integer with auto increment.
- **Velocidad:** the wind speed, no matter the measurement units.
- **Na:** this value can be 0 or 1, there are only a register with a value of 1 in this field, and indicates that this parameter has no value. The user only can create registers with this field with a value of 0, the na register is provided for the system by default.

Shooting position table

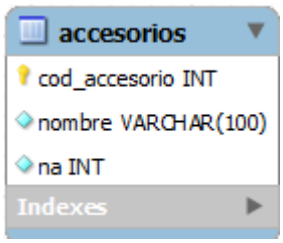
The table **posiciones** contains the different positions adopted in shooting, the fields are as follows:



- **cod_posicion:** the primary key, an integer with auto increment.
- **Descripcion:** a text string with the description of the position.

Accessories table

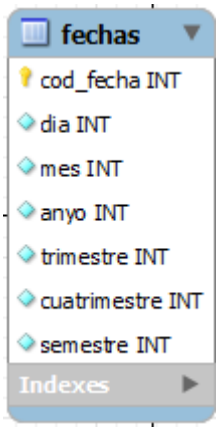
The table **accesorios** contains the different accesories added to the gun, the fields are as follows:



- **cod_accesorio**: the primary key, an integer with auto increment.
- **Nombre**: the name of the accesory.
- **Na:** this value can be 0 or 1, there are only a register with a value of 1 in this field, and indicates that this parameter has no value. The user only can create registers with this field with a value of 0, the na register is provided for the system by default.

Dates table

The table **fechas** contains a register for each single day with registered shots. The fields are as follows:



- **cod_fecha:** the primary key, an integer with auto increment.
- **Dia:** the day of month number.
- **Mes:** the month number.
- **Anyo:** the year number.
- **Trimestre:** the quarter number, from 1 to 4.
- **Cuatrimestre:** the fourth month period number, from 1 to 3.
- **Semestre:** the semester number, 1 or 2.

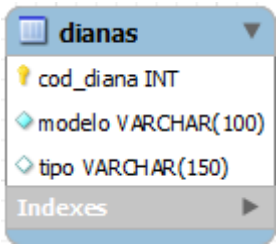
So it can be very hard to work with this table to filter the data, a calendar control is provided with the SDK that composes the filters for you and return them as a string that can be appended on the where clause of your SQL queries.

Target tables

These tables contain information of the target parameters.

Target models table

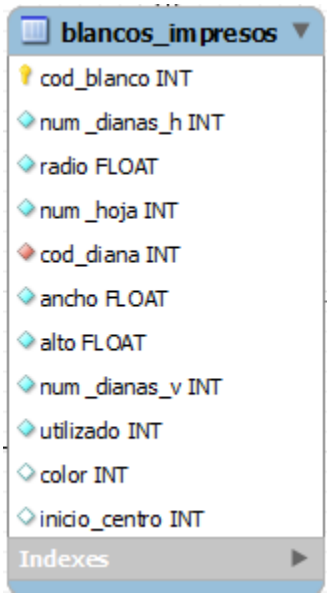
The table **dianas** contains the different target models provided by the program, and the new ones provided by the add-ons.



- **cod_diana:** the primary key, is an integer but nor an auto increment field. The values from 100 are for the extension targets.
- **Modelo:** a text string containing a mnemonic to identify the model.
- **Tipo**: This is the name of the type of the classes that implement the extension target models. For the models provided by the program is null.

Printed targets table

The table **blancos_impresos** contains all the printed targets and its configuration, the fields are as follows:



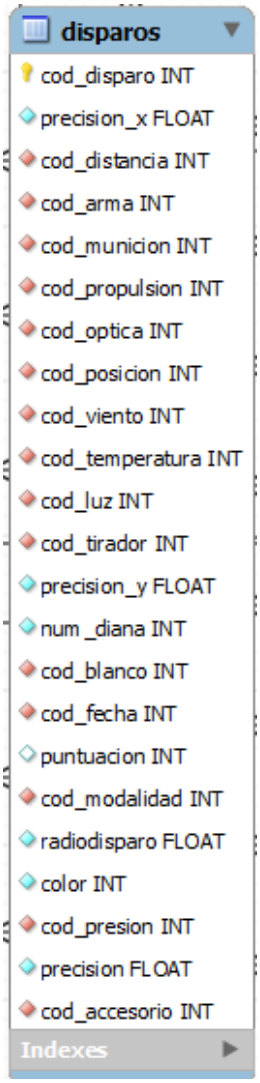
- **Cod_blanco:** the primary key, an integer with auto increment.
- **Cod_diana:** a foreign key to link with the target model in table **dianas**.
- **Num_dianas_h:** the number of targets in horizontal.
- **Num_dianas_v:** the number of targets in vertical.
- **Radio:** the radius of the target in millimeters.
- **Color:** an integer containing the ARGB value of the bulls eye color.
- **Inicio_centro:** The number of the target ring where the bulls eye begins.
- **Ancho:** the width, in inches/100, of the page containing the target.
- **Alto:** The height, in inches/100, of the page containing the target.
- **Num_hoja:** the number of the page, when various copies are printed.
- **Utilizado:** indicates if the target is not already used to shoot, with a 0 value, or if it is a completed target, with a 1 value.

Shot data tables

There are two tables that contain data on the shots, one containing data of each individual shot, and another containing aggregated data for an entire target.

Individual shot data table

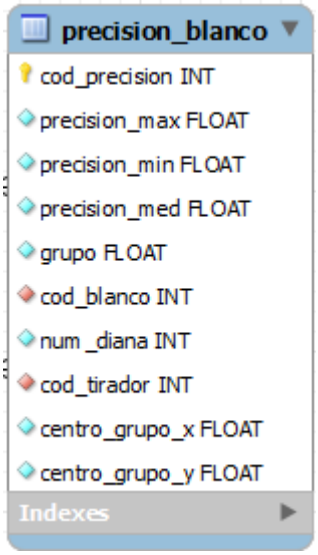
The table **disparos** contains a register for each single shot, the fields are as follows:



- **cod_disparo:** the primary key, an integer with auto increment.
- **Cod_distancia:** foreign key to the table of distances.
- **Cod_arma:** foreign key to the table of gun models.
- **Cod_municion:** foreign key to the table of ammo type.
- **Cod_propulsion:** foreign key to the table of propellant type.
- **Cod_optica:** foreign key to the table of optics.
- **Cod_popsicion:** foreign key to the table of shooting positions.
- **Cod_viento:** foreign key to the table of wind speed.
- **Cod_temperatura:** foreign key to the table of temperatures.
- **Cod_luz:** foreign key to the table of ambient light type.
- **Cod_tirador:** foreign key to the shooters table.
- **Cod_modalidad:** foreign key to the table of shot classes.
- **Cod_presion:** foreign key to the table of trigger weights.
- **Cod_blanco:** foreign key to the targets table, linking with the target to which the shot belongs.
- **Cod_accesorio:** foreign key to the accessories table .
- **Num_diana:** the target number in multi target sheets.
- **Cod_fecha:** foreign key to the table of dates.
- **Precision:** the distance in millimeters between the shot and the target centers.
- **Precision_x:** the position in millimeters of the center of the shot relative to the target center in the x (horizontal) axis (the center is 0).
- **Precision_y:** the position in millimeters of the center of the shot relative to the target center in the y (vertical) axis (the center is 0).
- **Radiodisparo:** the radius in millimeters of the shot.
- **Color:** the ARGB value of the color of the shot.
- **Puntuacion:** the score obtained with this shot. The value is 0 for targets with no scores or -1 to indicate a fail.

Aggregated target data table

The table **precision_blanco** contains aggregated data for all the shots of one shooter in a target. The fields are as follows:



- **cod_precision:** the primary key, an integer with auto increment.
- **Cod_blanco:** foreign key to the targets table.
- **Num_diana:** number of the target in multi target sheets.
- **Cod_tirador:** foreign key to the table of shooters.
- **Precision_max:** the maximum distance to the target center obtained, in millimeters.
- **Precision_min:** the minimum distance to the target center obtained, in millimeters.
- **Precision_med:** the average distance to the target center, in millimeters.
- **Grupo:** the size in millimeters of the group of shots.
- **Centro_grupo_x:** the horizontal coordinate of the center of the group circle, relative to the target center (0, 0), in millimeters.
- **Centro_grupo_y:** the vertical coordinate of the center of the group circle, relative to the target center (0, 0), in millimeters.

The group circle is not the real center of the group, but the center of the circle that has the farthest two shots in opposite positions relative to their diameter, as displayed by the program.

Parameter cards tables

There are some tables dedicated to store the default parameters and shooter lists defined by the user.

Default shooting parameters card table

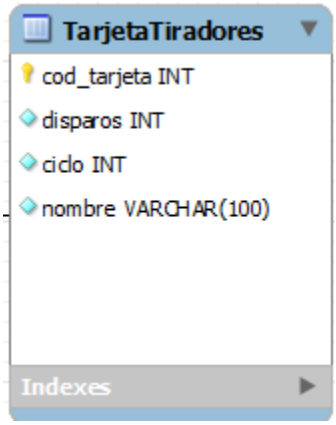
The table **TarjetaTiro** contains a set of default parameters to speed up the selection of shooting parameters by the user. The fields are as follows:

- **cod_tarjeta:** the primary key, an integer with auto increment.
- **cod_posicion:** foreign key to the shooting positions table, can be null.
- **Cod_luz:** foreign key to the ambient light table, can be null.
- **Cod_optica:** foreign key to the optics table, can be null.
- **Cod_propulsion:** foreign key to the propellants table, can be null.
- **Cod_temperatura:** foreign key to the temperature table, can be null.
- **Cod_municion:** foreign key to the ammo type table, can be null.
- **Cod_viento:** foreign key to the wind speed table, can be null.
- **Cod_distancia:** foreign key to the distance table, can be null.
- **Cod_arma:** foreign key to the gun model table, can be null.
- **Cod_presion:** foreign key to the trigger weight table, can be null.
- **Cod_modalidad:** foreign key to the shot class table, can be null.
- **Cod_accesorio:** foreign key to the accessories table, can be null.
- **Nombre:** name of the shooting card.

Note that the shooter is not listed in these cards. The shooters are selected using the shooter list cards.

Shooter list cards table

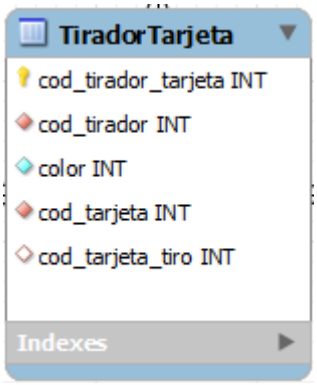
In order to create lists of shooter for the multi shooter mode, there is the table **TarjetaTiradores** that contains the headers of each of these lists. The fields are as follows:



- **cod_tarjeta:** the primary key, an integer with auto increment.
- **Nombre:** the name of the shooter list.
- **Disparos:** the default number of shots by shooter.
- **Ciclo:** A default value for the cycle mode control of the shooting gallery.

Shooter item table

For each shooter in the shooter list, there is a register in the table **TiradorTarjeta**, the fields are as follows:



- **Cod_tirador_tarjeta**: the primary key, an integer with auto increment.
- **Cod_tarjeta:** foreign key to the list of shooters table.
- **Cod_tarjeta_tiro:** foreign key to the shooting parameters card table. If not null, these parameters apply in the turn of the shooter.
- **Cod_tirador:** foreign key to the shooters table.
- **Color:** the ARGB value with the color of the shots assigned to the shooter.

Composite filters tables

Finally, there are two tables containing the data for the composite filters designed by the user. Because these tables contain data referring to classes and properties of the code, they are intended for internal use only. To manage the composite filters, you can use instead of them the classes provided with the SDK, which return text strings with the filter conditions that can be used to compose the where clause of your queries.

The kernel of E-Sniper SDK

The communication between your software and the program E-Sniper is accomplished by means the **E-Sniper.AddOns.dll** library.

This library defines a series of contracts in the form of interfaces that must be meet by the classes to be linked with the program for be inserted as modules in their infrastructure. The only thing you need to do is create the public classes that implement these interfaces and E-Sniper will search in the library to find and link them to the program.

For the moment, there are only three options to extend the program, create new forms, new target models and add new camera controllers.

E-Snipe.AddOns.dll also provides a set of utility classes to simplify the management of the program objects and database access.

You must add a reference to this library in all your projects. There is no need to distribute the library, as it is part of the standard installation of E-Sniper.

E-Sniper add-ons interfaces

There are tree interfaces that you can use to link new classes with the program, there are listed below. All interfaces reside in the namespace **E_Sniper.AddOns.Interfaces**.

IConnectionProvider interface

This interface is intended to be implemented for all the classes that must use or provide a database connection. By means of this interface, the program passes to your classes the actual database connection selected by the user.

You must provide also this database connection back to some of the utility classes of the SDK library.

```
namespace E_Sniper.AddOns.Interfaces
{
    public interface IConnectionProvider
    {
        DbConnection Connection { get; set; }
    }
}
```

The only member of this interface is a property to get and set the actual database connection. As the user can change this connection at any moment, you must respond properly to these changes.

IExtensionForm interface

This is the interface you must implement if you want to add new forms to the program. If your form also needs database support, you must also implement IConnectionProvider.

```
namespace E_Sniper.AddOns.Interfaces
{
    public interface IExtensionForm
    {
        string UID { get; }
        string MenuOption { get; }
    }
}
```

All the forms in the E-Sniper program have only one instance open at once, so they need a unique identifier to be found by the program to open or show it when the user requests them.

This is accomplished by providing this unique identifier in the **UID** property. You can use a GUID stored in the resource file, since these are statistically unique. If you prefer, you can provide any string that you consider that is unique instead.

The property **MenuOption** returns the text of the menu option that will be appended in the AddOns menu of the program to launch your form. Since the E-Sniper program is multi language, it is a good practice return a localized string at least in English and Spanish, depending of the actual language selected by the user. You can view samples of how to do this in the code samples accompanying the SDK documentation.

Of course, all the classes that implement this interface need to be also instances of the Form class of the framework.

IEsniperCamera interface

If you want to extend the cameras available to the program with your own camera controller, you must create a class that implement this interface. The camera will appear in the list of cameras in the shooting gallery once installed the module.

```
namespace E_Sniper.AddOns.Interfaces
{
    public interface IEsniperCamera
    {
        int Width { get; set; }
        int Height { get; set; }
        int Fps { get; set; }
        bool NeedsParameters { get; set; }
        void ShowParameterDialog();
        void ShowCameraConfiguration();
        void Start();
        void Close();
        event Action<Bitmap> OnNewImage;
        void ReleaseEvents();
    }
}
```

The **Width** and **Height** properties are for the image size of the captured images. E-Sniper uses always a resolution of 640x480 pixels, so you can assume this resolution and do not do anything in the body of these properties.

The same is valid for the property **Fps**, which stands for the frame capture rate. E-Sniper always passes the value of 10 frames per second to this property, so you can assume this value by default.

The property **NeedsParameters** is intended for indicate if there is the need to request initialization parameters to the user, like the URL or IP of the camera, the login data, or any other parameters needed to start operating with the camera.

If there is the need to initialize the camera before using it, the first thing E-Sniper does is to call the **ShowParameterDialog** member function. You must show a modal dialog box which for the user to enter these initialization parameters. This function is called every time you return true in the **NeedsParameters** property, so you must return false if there are no need to initialize the camera again every time.

In the **ShowCameraConfiguration** member function you must show a dialog box with the controls of the camera settings. This dialog box cannot be modal, as the camera are capturing images at the same time.

In response to the **Start** member function, you must initiate the image capture, one frame at time. The images are passed to the program using the **OnNewImage** event, which is a delegate with a unique parameter of **Bitmap** type. The delegate is thread safe, so you can instantiate a thread to capture the images. You can use also one of the various timer options provided by the framework. We recommend to not use the timer in the System.Windows.Forms namespace, as it can interfere with the timer used in the shooting gallery module. Anyway, you must not block the program flow in this function, or the program can be malfunctioning.

The **Stop** member function finishes the camera operation.

The **ReleaseEvents** member function is used to initialize to null the **OnNewImage** event. This simplifies the camera management to E-Sniper.

ITargetBase interface

This is the interface that provides services to the extension target models. It acts like a base class for your objects.

```
namespace E_Sniper.AddOns.Interfaces
{
    public interface ITargetBase
    {
        IExtensionTarget Extension { get; set; }
        int TargetsHorizontal { get; set; }
        int TargetsVertical { get; set; }
        float Radius { get; set; }
        KeyValuePairES<bool, string> PrintNumbers { get; set; }
        KeyValuePairES<bool, string> ExternalGuides { get; set; }
        float ExternalGuidesThick { get; set; }
        KeyValuePairES<bool, string> InternalGuides { get; set; }
        float InternalGuidesThick { get; set; }
        KeyValuePairES<bool, string> Border { get; set; }
        float BorderThick { get; set; }
        float RingThick { get; set; }
        KeyValuePairES<Color, string> CenterColor { get; set; }
        int CenterStart { get; set; }
        void DrawFramePoints(Graphics gr);
        void PageSize(out int mmx, out int mmy, out Size border);
    }
}
```

The properties exposed by the interface are as follows:

- **Extension**: get or set the object that implements the target model. You do not should manage this property.
- **TargetsHorizontal**: gets or sets the number of targets in horizontal. If your model is a single target, you must pass always 1. In models with multiple targets, the minimum value is 2, or an exception will be thrown.
- **TargetsVertical**: gets or sets the number of targets in vertical. If your model is a single target, you must pass always 1. In models with multiple targets, the minimum value is 2, or an exception will be thrown.
- **Radius**: The radius of the target in millimeters. Only values that fit in the sheet are allowed. You do not need to deal with the calculations involved in this restriction, simply pass the value to the container and it does for you.
- **PrintNumbers:** indicates, for multiple targets page, if you must print the target number or not.

- **ExternalGuides**: if your model needs some kind of guides external to the target, use this property to indicate if they should be drawn or not.
- **ExternalGuidesThick:** The thickness in millimeters of the external guides, if they should be printed.
- **InternalGuides**: if your model needs some kind of guides internal to the target, use this property to indicate if they should be drawn or not.
- **ExternalGuidesThick:** The thickness in millimeters of the internal guides, if they should be printed.
- **Border**: use this property your model has the option to draw an extra border around the target, to allow the user to select draw it or not.
- **BorderThick**: The thickness of the extra border, in millimeters.
- **RingThick**: The thickness of the lines of the target, in millimeters.
- **CenterColor:** The color of the target bullseye.
- **CenterStart:** The number of the ring where the bullseye begins.

There are also two methods exposed by this interface:

- **DrawFramePoints**: If your model is a single target, four frame points should be drawn to allow the user to indicate the target position in the camera image. These points must be symmetrical respect to the center of the target, one in each corner of a rectangle exterior to the target. You must pass to the method the **Graphics** object that you are using to draw the target. Do not call this method if your model is a multi target one, as in these models the frame is performed using the centers of the targets in the corners of the sheet.
- **PageSize:** Call this method to get the size of the printable surface of the sheet, as the user can select different page sizes. You obtain three measures, the width of the printable surface in millimeters, the height in millimeters and the size of the borders in millimeters.

IExtensionTarget interface

You must implement this interface in all the classes of extension targets, in order to be linked with the E-Sniper program.

The public properties of your target objects are exposed to the user in a **PropertyGrid** control, so all the properties that you do not want the user can view and change must be decorated with the **Browsable**(false) attribute of the System.ComponentModel namespace.

You can use the localized version of the attributes **DisplayName** and **Description** provided in the source code samples to decorate your exposed properties and show the user these text strings in English or Spanish according the user language selection.

All the properties in this interface must be managed by the **ITargetBase** container object, to let it validate the data. Never use local variables to store their values, instead, get the value from the container every time you need to use it, and pass the value entered by the user without processing to the container, acting like a bridge between the user and the application.

Do not use colors in your target other than the returned by the **CenterColor** property, this may cause that the shot detection fail and make your target unusable, neither use shades of gray inside the target. The target surface must be as clear and simpler as possible.

The definition of the interface is as follows:

```
namespace E_Sniper.AddOns.Interfaces
{
    public interface IExtensionTarget
    {
        string Tip { get; }
        string Mnemonic { get; }
        bool DebugMode { get; }
        float DebugAngle { get; }
        int TargetsHorizontal { get; set; }
        int TargetsVertical { get; set; }
        float Radius { get; set; }
        bool AutoFrame { get; }
        KeyValuePairES<bool, string> PrintNumbers { get; set; }
        KeyValuePairES<bool, string> ExternalGuides { get; set; }
        float ExternalGuidesThick { get; set; }
        KeyValuePairES<bool, string> InternalGuides { get; set; }
        float InternalGuidesThick { get; set; }
        KeyValuePairES<bool, string> Border { get; set; }
        float BorderThick { get; set; }
        float RingThick { get; set; }
        KeyValuePairES<Color, string> CenterColor { get; set; }
        int CenterStart { get; set; }
        List<RectangleF> BlindAreas { get; }
        List<PointF> TargetCenters { get; }
        RectangleF Frame { get; }
        void PrintTarget(Graphics gr, int number);
        int Score(PointF center, float calibre);
    }
}
```

The properties in this interface are:

- **Tip**: This is a short text to show to the user when the mouse is over the small picture of the target in the list of available targets of the target module. If possible, return a version in English or Spanish according with the current language. Do not expose this property to the user, decorate it always with the **Browsable**(false) attribute.
- **Mnemonic**: This is the text stored in the **modelo** field of the **dianas** table and the title printed in the top left corner of the page. Do not expose this property to the user. This string is not localized, and must be a constant value.
- **DebugMode:** if you return true in this property, the program will use the target in debug mode in the shooting gallery. In this way, the program will simulate a line of shots crossing the center of the target by at regular intervals, so that you can assess whether your function to calculate the score performs calculations correctly. The mechanism is the same as with real shots, you print the target, frame it and then press the shoot button and the shots will be drawn and the score calculated for each one. The target is saved in the database so you can check the scores in the module shots and targets query, so if you want to repeat the test print off another target of the same features using a virtual printer, as PDF or XPS printers.
- **DebugAngle:** If your target is not circular, you can vary the angle of the line of virtual shots of the debugging mode returning the angle in degrees.
- **TargetsHorizontal**: gets or sets the number of targets in horizontal. If your model is a single target, you must return always 1. In models with multiple targets, the minimum value is 2, or an exception will be thrown.

- **TargetsVertical**: gets or sets the number of targets in vertical. If your model is a single target, you must return always 1. In models with multiple targets, the minimum value is 2, or an exception will be thrown.
- **Radius**: The radius of the target in millimeters. Only values that fit in the sheet are allowed. You do not need to deal with the calculations involved in this restriction, simply pass the value to the container and it does for you.
- **AutoFrame:** Indicates if your target is valid to auto-frame.
- **PrintNumbers:** in multiple targets pages, indicates if you must print the target number or not.
- **ExternalGuides**: if your model needs some kind of guides external to the target, use this property to indicate if they should be drawn or not. The SDK provide a customized editor for this property, it is a KeyValuePairES to allow you to show a translation of the Boolean value to the user, by example, yes/no.
- **ExternalGuidesThick:** The thickness in millimeters of the external guides, if they should be printed..
- **InternalGuides**: if your model needs some kind of guides internal to the target, use this property to indicate if they should be drawn or not. The SDK provide a customized editor for this property, it is a KeyValuePairES to allow you to show a translation of the Boolean value to the user, by example, yes/no.
- **ExternalGuidesThick:** The thickness in millimeters of the internal guides, if they should be printed.
- **Border**: use this property your model has the option to draw an extra border around the target, to allow the user to select draw it or not.
- **BorderThick**: The thickness of the extra border, in millimeters.
- **RingThick**: The thickness of the lines of the target, in millimeters.
- **CenterColor:** The color of the target bullseye. You must always use the property editor provided by the SDK to edit this property. Not all colors are allowed, and this editor limits the selection at these ones that work fine.
- **CenterStart:** The number of the ring where the bullseye begins.
- **BlindAreas:** if your target draws some figures outside his shooting surface, these can produce false shot detection. Use this property to return a list of rectangular areas to be excluded of the shot detection. By example, in multiple targets, the number of the targets must be excluded. The coordinates and sizes are in millimeters. You can return null in this property.
- **TargetCenters:** You must return a list of the coordinates of the center of all the targets in the sheet, ordered by target number. The targets must be numbered from left to right and from top to bottom. The coordinates are in millimeters.
- **Frame:** Return a rectangle with the coordinates of the four points used to locate the targets in the framing operation. In single targets, the center of this rectangle is the center of the target, in sheets with multiple targets, the corners must coincide with the coordinates of the centers of the targets of the four corners of the page. The coordinates are in millimeters.

Keep in mind that not all of these values are stored in the database, and cannot be reproduced in the drawings of the target, so try to not use other custom properties than these provided by the interface, since their values will be lost.

You must implement also two methods:

- **PrintTarget**: In this method, you must draw the target in the **Graphics** passed in the gr parameter. This Graphics object is scaled before passing it to you, so you must use always millimeters as the units of the coordinates of the points in your drawing, no matter which is the destination of the picture. See the code

samples to view a single and multiple target implementation of this method. The parameter number is simply the number of the page, to show it in the top left corner if you want.

- **Score:** As you are the only one who can know how to score the shots, the system calls this method to calculate this score. The parameters are the center of the shot, in millimeters, relative to the center of the target, so you do not need to know on which of the targets on the page is the shot. The other parameter is the radius of the orifice, also in millimeters.
  You must return a number between 1 and 10, or -1 if the shot is a fail. If your target not uses scores, simply return 0.

Override the **ToString** method to provide a description for all the set of properties in the PropertyGrid editor.

Utility classes and components provided by the E-Sniper.AddOns.dll library

There are many classes and controls that you can use in your add-on to simplify the E-Sniper data management.

CalendarPeriod enumeration

This enumeration defines the various types of date periods managed by the calendar control; they correspond with the fields of the table of dates. It is defined in the **E_Sniper.AddOns.Controls** namespace.

- **Day:** the period is a single day.
- **Month:** the period is a complete month.
- **Year:** the period is a year.
- **Quarter:** the period is a quarter.
- **FourMont:** the period is a four month period**.**
- **Semester:** the period is a semester.

CalendarElement class

Each instance of this class represents a time period of the calendar. It can be of anyone of the period types listed in the CalendarPeriod enumeration, but all the periods managed by the calendar control at a given moment are of the same type. It is defined in the **E_Sniper.AddOns.Controls** namespace..

To create an instance of this class, you can use anyone of these constructors:

- `public CalendarElement(int day, int month, int year)`
  creates an instance of type Day.
- `public CalendarElement(int month, int year)`
  creates an instance of type Month.
- `public CalendarElement(int year)`
  creates an instance of type Year.
- `public CalendarElement(int year, int division, CalendarPeriod ctype)`
  creates an instance of type Quarter, FourMonth or Semester, depending on the ctype parameter. The division parameter indicates the number of the quarter, four month or semester, starting in 1.
- `public CalendarElement(CalendarElement copia)`
  creates a copy of the given CalendarElement.

The public properties exposed by this class are the following:

- `CalendarPeriod Ctype:` gets the type of period of this element.
- `int Day:` Gets the day component of the date.
- `int Month:` Gets the month component of the date.
- `int Year:` Gets the year component of the date.
- `int Quarter:` Gets the quarter component of the date.
- `int FourMonth:` Gets the four month component of the date.
- `int Semester:` Gets the semester of the date.

You can use the following operators with the CalendarElement objects:

- `int operator -(CalendarElement e1, CalendarElement e2)`
  Calculates the difference in CalendarPeriod units between the e1 and e2 calendar elements. The two elements must be of the same type. By example, if there are months, it returns the difference in months of the two periods, if there are days, the difference in days, and so on.
- `CalendarElement operator +(CalendarElement e1, int n)`
  Calculates the result of adding the n number of units of calendar periods (depending of the type of the period of the e1 CalendarElement). By example, if e1 is a month period, n stands for months, and the result is n months after the e1 period.
- `CalendarElement operator -(CalendarElement e1, int n)`
  Same as the preceding, but subtracting n calendar periods.

The public member functions provided by this class are the following:

- `DateTime ToDateTime()`
  Converts the CalendarElement to a DateTime object.
- `bool Equals(CalendarElement otro)`
  Compares the CalendarElement with another and returns true if they represent the same period. This is the implementation of the IEquatable interface. If the two elements are of different types, an exception is thrown.
- `int CompareTo(CalendarElement otro)`
  This is the implementation of the IComaparable interface. If the elements are of different types, an exception is thrown.
- `string ToShortString() and string ToString()`
  Returns a string representing the time period, in short date format mode.
- `bool Belongs(CalendarElement elm)`
  If the CalendarElement elm is between the start and end of this period element, it returns true.
- `void Convert(CalendarPeriod CType)`
  Converts the calendar period to the given CalendarPeriod type. This can be problematic on converting periods of type quarter or four month.

There are also some functions to generate filters for use in SQL queries:

- `string ToString(bool l)`
  Returns a equals filter for use in the where clause of a SQL query. The parameter as no meaning and is used only to differentiate of the standard ToString function that returns a representation of the date.
- `string ToString(bool greater, bool equal)`
  Composes a greater than, less than, greater or equal than or less or equal than filter for use in the where clause of a SQL query.

CalendarRange class

This class represents a range of dates of the same CalendarPeriod type. It is defined in the **E_Sniper.AddOns.Controls** namespace.

To create an instance of this class use the following constructor:

```
public CalendarRange(CalendarElement e1, CalendarElement e2)
```

The two calendar elements must be of the same type of period. The constructor selects automatically the start and end period depending of the dates represented by the parameters.

There are three public properties in this class:

- `CalendarElement Start:` Gets the start time period of the range.
- `CalendarElement End:` Gets the ending time period of the range.
- `int Range:` Gets the difference in period units of the start and ending time periods (by example, if there are of year type, returns the difference in years).

With this function you obtain a filter string to append in a where clause of a SQL query with the start and end dates of the range:

```
string ToString()
```

Calendar control

In the namespace **E_Sniper.AddOns.Controls** there is a calendar control that you can use to allow the user to select dates to filter queries. The class name of this control is **Calendar**. The selection in this control can consists on a list of elements of type **CalendarElement** or a single element of type **CalendarRange**, depending of the type of selection made by the user.

The public properties of this control are as follows:

- `CalendarPeriod PeriodType:` Gets the type of time period selected currently in the calendar. If you change the value of this property, the control loses the current selection and displays the new type of date period.
- `string WhereClause:` Gets a string composed with the current selection as a filter that you can use in the where clause of a SQL query.
- `CalendarRange SelectionRange:` if the type of selection made by the user is a range of dates, gets this range. If the selection consists on a set of dates, this property returns null.

If you want to be notified when the user changes the current selection, you can subscribe to the following EventHandler event:

```
EventHandler OnChange
```

You can clear the current selection programmatically using the member functions:

- `void InitializeSelection()`: Clear the current selection.
- `void InitializeSelection(CalendarElement sel)`: Clear the current selection and sets the sel CalendarElement as the current selection, changing the period type to the type of the sel element.

To manage the selection of a set, not necessarily continuous of dates of the same type of period, you can use the following methods:

- `List<CalendarElement> GetSelection()`: Returns the current selection made by the user in the calendar. If the selection is a range of dates, it returns an empty list. In this case, you can check the value of the SelectionRange property to get the current selection.
- `void SetSelection(List<CalendarElement> value)`: Sets the current selection to the given list of CalendarElement objects. All the items in the list must be of the same type of period, or an exception is thrown.

Finally, if you want to get a textual representation of the selected dates in the calendar, you can use the **ToString** method, which returns the initial and final dates of the selection separated by suspension points if the selection is a list, or by a hyphen if it is a continuous range of dates.

The KeyValuePairES generic class

In the namespace **E_Sniper.AddOns.Utils** it is defined the class **KeyValuePairES**, which is a version of the generic KeyValuePair class of the framework used by the utility classes that deal with PropertyGrid editor controls. This version implements the IComparable and IEquatable interfaces. This class is also marked as serializable.

The prototype is:

```
public class KeyValuePairES<C, V>
```

Being C the type of the Key and V the type of the value.

The constructor is as follows:

```
public KeyValuePairES(C clave, V valor)
```

You can get or set the **Key** and the **Value** using the properties of the same name.

The **ToString** method returns the ToString version of the Value.

The **CompareTo** method of the IComparable interface compares the value of this object with the value of another, using the CompareTo method if they are IComparable also, or comparing it as string if not.

The **Equals** method of the IEquatable interface returns true if this object and the argument have equal keys and values.

The PropertyGrid control

The E-Sniper application uses the PropertyGrid control provided by the framework to edit sets of properties, so the E-Sniper.AddOns.dll library defines a set of classes for ease you the use of this control in order to maintain a homogeneous user interface.

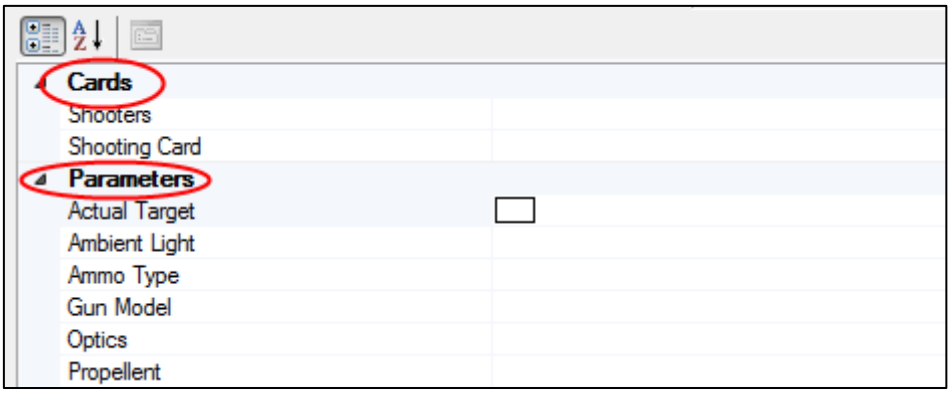
The PropertyGrid control is not added by default to the VisualStudio tool box, so if you do not see it, you must install it by clicking on the tool box with the right mouse button and selecting the select elements option. A dialog box will open and you must select the .NET framework components tab and then find the PropertyGrid component and mark it to install.

Edit the public properties of an object with this control is easy, you must use attributes to define the behavior of each property in the grid, and then pass your object to the control by means of the property **SelectedObject**.

The standard attributes used to customize the display are defined in the System.ComponentModel namespace.

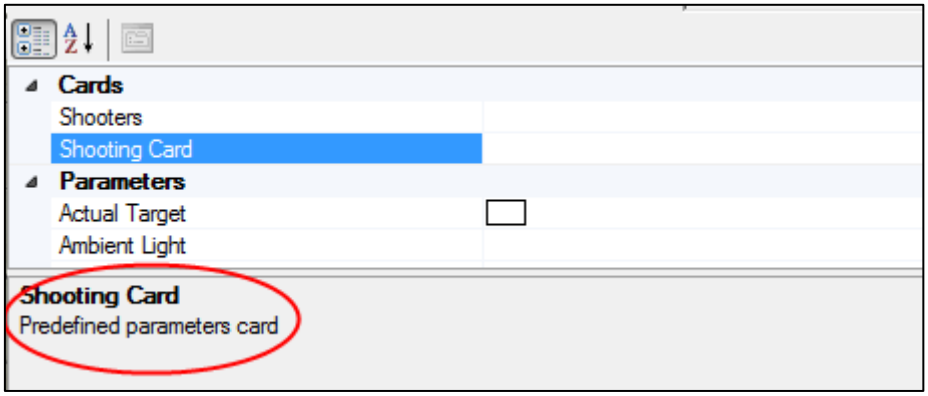
You can use the **[Browsable(false)]** attribute to prevent a property to be edited by the grid.

With the **[Category(string)]** attribute you can separate the properties in groups.

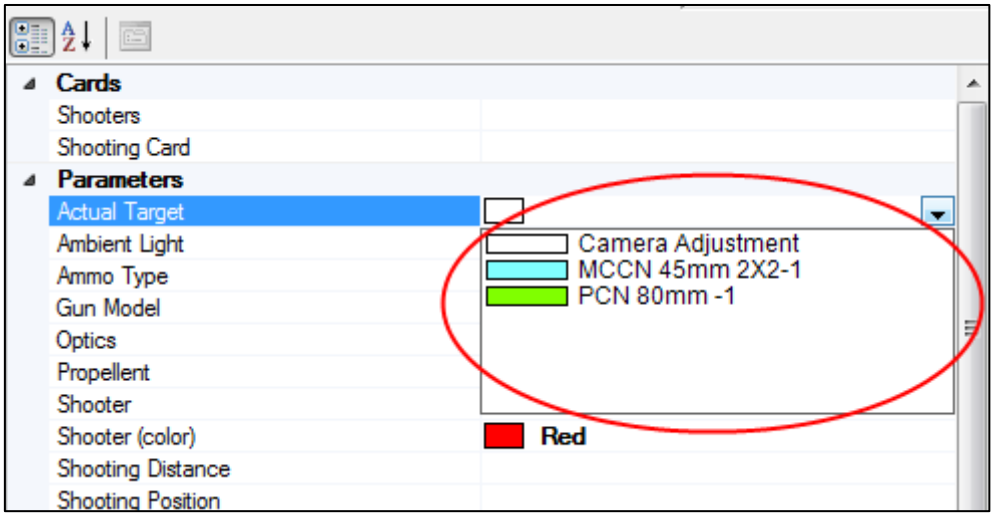


With the **[DisplayName(string)]** attribute you can set the name of the property in the grid.

With the **[Description(string)]** attribute you can provide a description to the user when the property is selected.



Finally, you can customize the property editor by subclassing the UITypeEditor class defined in the System.Drawing.Design namespace and using the **[Editor(Type, Type)]** attribute.



In the SDK sample code you can find a localized version of these attributes, and you can use it on your projects to support English and Spanish versions of the property names and descriptions.

Property editors provided by the E-Sniper.AddOns.dlll library

In the namespace E_Sniper.AddOns.PropertyEditors there are some property editors that you can use to allow the user to select values from the database.

- `AccesoriesEditor`: Shows a dropdown list with all the gun accessories in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `GunModelEditor`: Shows a drop down list with all the gun models in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `DistanceEditor`: Shows a drop down list with all the distances in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `LightEditor`: Shows a drop down list with all the ambient light in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `ClassEditor`: Shows a drop down list with all the shot classes in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `AmmoEditor`: Shows a drop down list with all the ammo types in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `OpticsEditor`: Shows a drop down list with all the optics in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.

- `PositionEditor`: Shows a drop down list with all the shooting positions in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `TriggerWeightEditor`: Shows a drop down list with all the trigger weight values in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `PropellantEditor`: Shows a drop down list with all the propellant types in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `TemperatureEditor`: Shows a drop down list with all the temperature values in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `ShooterEditor`: Shows a drop down list with all the shooter names in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `WindSpeedEditor`: Shows a drop down list with all the wind speed values in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `FilterEditor`: Shows a drop down list with all the composite filter names in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES<int, string>, being the key the code of the register and the value the display value.
- `TargetTypeEditor`: Shows a drop down list with all the different target models in the current database. The object passed to the property grid must implement the IConnectionProvider interface, and the property must be of type KeyValuePairES< KeyValuePairES<string, float>, string>. The inner KeyValuePairES<string, float> contains the target model in the key and the target radius in the value, the value of the outer KeyValuePairES is a composed string with the model and the radius.
- `CalendarEditor`: Shows a Calendar control to allow the user to select dates, the property must be of type KeyValuePairES<int, Calendar>.
- `TargetColorEditor`: Allows to select the color of the center of the target between the set of the permitted colors. The property must be of type KeyValuePairES<Color, string>. Only the Key is used by this editor.
- `BooleanEditor`: Allows edit Boolean properties providing a best descriptor of the meaning of the value than the standard true/false, by example yes/no. The property must be of type KeyValuePairES<bool string>, being the value the translation to natural language of the truth value of the property.

You can find samples of use of all of these property editors in the sample code accompanying the SDK.

Manage composite filters

In order to ease the management of composite filters created by the user, the E-Sniper.AddOns.dll library provides a class that hides all the complexity of the filter tables and gets you a string ready to append in the where clauses of your SQL queries. This class resides in the **E_Sniper.AddOns.Database** namespace.

CompositeFilter class

In order to use the instances of this class, you must provide an IConnectionProvider object to access to the current database. Use the following constructor:

`public CompositeFilter(int codfilter, IConnectionProvider prov):` the parameter codfilter stands for the code of the filter, the primary key of the filters table.

Alternatively, you can provide the IConnectionProvider by means of the static property `static IConnectionProvider ConnectionProvider` and use the following constructor to instantiate the objects:

```
public CompositeFilter(int codfilter)
```

With the following properties, you can get information on the filter components:

- `List<string> Fields:` each string of the list has the format “esniper.<table name>.<field name> as <alias>”. The alias is assigned by the CompositeFilter class; it is composed with the name of the field and the code of the filter, in order to allow the user use more than one filter with the same fields in the same query. You can use this list to construct the select field list in the SQL query.
- `List<string> FieldsOG:` each string in this list has the format “esniper.<table name>.<field name>”. There is only one string for each different field in the filter, and you can use this list to construct group by and order by clauses.
- `List<string> Links:` each string in this list has the format “join esniper.<table name> on esniper.disparos.<foreign key name>=esniper.<table name>.<field name>”. These join clauses are for the filter conditions that use an operator different to =, like < or >=, where there is a range of values that can satisfy the condition instead a fixed value, that can be filtered without linking with the value table, simply comparing the primary key directly on the shot table. You can use this list to construct the from clause of your SQL queries.
- `string Conditions:` Gets a string with all the filter conditions chained, ready to append to your where clause. See the sample code accompanying this documentation for a sample of use.

When you execute a query composed with more than one composite filter, it is possible that you want separate the data obtained by applying each one of the filters. For this there is the following method of the CompositeFilter class:

```
bool SatisfyCondition(List<KeyValuePairES<string, string>> fields)
```

Construct the fields list with KeyValuePairES objects whose key is the alias of the field and the value the value to compare to. The function returns true only if the values satisfy the filter conditions.

By example, suppose that you have executed the query and filled a DataSet with the data. You can test if a row is the result of apply one composite filter by constructing a KeyValuePairES list composed with the column name (which is the alias of the field) and the value in the row for that column, then pass this list to each CompositeFilter by means of this function until you obtain a true as the result.

All the columns that not belong to the filter are ignored, so you no need to worry on selecting the columns that you put in the list, add all columns always.

Make your add on installable

In order to allow E-Sniper install your add-ons, you must provide an xml file defining the list of files to copy and which of these files are code modules.

The name of the file must be **setup.xml**.

XML elements of the setup file

Structure elements

The name of the root element of the document is **addon**.

The **addon** root element contains two container elements:

- **files** element, is mandatory and contains a list of elements **file,** each representing one of the files to copy.
- **addons** element, is optional and contains a list of elements **module**, one for each assembly to link with E-Sniper.

The reason why the addons element is optional is that you can create an add-on with only a set of files to copy but not code modules, by example, to replace the wav files of the default locutions of the program. In this case, you only need to specify the list of files to copy.

File elements

Each element of type **file** has an attribute **location** that indicates the destination location, relative to the root directory where E-Sniper is installed. If you want to copy the file in the root directory, specify **root** as the value of the attribute. In other case, specify the complete route of the subfolder. If the directory does not exist, they will be created before copy the file.

The E-Sniper application has the following directory structure:

- \ : the root installation folder, contains the executables, configuration files and dll assemblies.
  - **en**: English localized resources.
  - **es**: Spanish localized resources.
  - **Ayuda**
    - **en:** English help files.
    - **es:** Spanish help files.
  - **DB:** local database files
    - **Scripts:** SQL scripts to create databases in the server.
    - **Utiles:** Empty local database.
  - **Sonido**
    - **en:** English locution wav files.
    - **es:** Spanish locution wav files.

The inner text of the element is the route, relative to the directory containing the add-on files, of the file to copy.

This is a sample of **file** element:

```
<file location="en">en\SDK.Demo.AddOn.resources.dll</file>
```

This line copies the file SDK.Demo.AddOn.resources.dll, located in the subfolder en, to the folder <E-Sniper root>\en.

Module elements

Each element of type **module** must contain an element **assembly** containing the name of the add-on library:

```
<assembly>SDK.Demo.AddOn.dll</assembly>
```

It can contain also an element **description** with a description, contained in a **name** element, to show to the user in the Add-Ons manager dialog. You can provide the description either in English, Spanish or the two languages by using the **lang** attribute:

```
      <description>
        <name lang="es">Cámara IP netwave y distribución de disparos</name>
        <name lang="en">Netwave IP camera and shot distribution</name>
      </description>
```

Sample setup.xml file

Finally, this is the complete setup.xml file used to install the SDK.Demo.AddOn sample module:

```
<addon>
  <files>
    <file location="root">SDK.Demo.AddOn.dll</file>
    <file location="en">en\SDK.Demo.AddOn.resources.dll</file>
    <file location="es">es\SDK.Demo.AddOn.resources.dll</file>
  </files>
  <addons>
    <module>
      <assembly>SDK.Demo.AddOn.dll</assembly>
      <description>
        <name lang="es">Cámara IP netwave y distribución de disparos</name>
        <name lang="en">Netwave IP camera and shot distribution</name>
      </description>
    </module>
  </addons>
</addon>
```