Smart PropertyGrid.Net

Programmer's Guide

(Written for version 2.2.x)

CONTENTS

1 Introduction

1.1 What is Smart PropertyGrid.Net?

Smart PropertyGrid.Net (SPG) version 2.1 is a complete, user-friendly and flexible replacement for Microsoft PropertyGrid. It provides a complete framework based on a solid object design.

Microsoft PropertyGrid (MSPG) - commonly referred to as "PropertyGrid" – is available in various integrated development environments, e.g. Visual Studio 2005. It has great potential for use in client applications, but was designed as a developer tool, rather than for end-users.

Rather than modify the base PropertyGrid class, VisualHint decided to create a flexible framework from scratch. The result is **Smart PropertyGrid.Net**.

The product evolves fast and your feature requests are studied carefully for later inclusion. This makes Smart PropertyGrid.Net the first in its category.

1.2 Features in version 2.1

Version 2.0 was a major milestone, definitively superseding MSPG in any client application. Version 2.1 builds on this by adding support for .Net 1.1.

Main features

- ☑ Built with the end-user in mind, e.g. supports checkboxes for booleans (with or without text), rather than a combobox.
- ☑ Flexible approaches to populating the grid ranging from the legacy *SelectedObject(s)* through to new Append and Insert calls.
- ☑ Inplace editors appear in the Property value column, rather than as dropdowns. These include checkboxes, radiobuttons, up/down buttons, trackbars, progress bars, color editors, and many more.
- ☑ Dynamic modification of Properties. Expand, disable and hide Properties and their counterparts at will. Change fonts and colors. Set possible values and read-only state at runtime.
- ☑ Easy porting of code from the MSPG to SPG, which is globally compatible with its predecessor. For example, in both SPG and MSPG, *TypeConverter*, *ICustomTypeDescriptor*, and *UITypeEditor* classes return Property information.
- ☑ Customizable appearance. The whole grid and each individual property can be made to match your application.

☑ Sort Properties and validate their values as required, without displaying annoying popups.

Known limitations under .Net 1.1:
-No masked textbox.
-Not possible to use the attribute PasswordPropertyTextAttribute.

1.3 Requirements

Smart PropertyGrid.Net is a library written in C# and supports .Net framework version 2.0 and 1.1.

Requirements

☑ .Net framework version 1.1 or 2.0.

☑ Visual Studio 2003, Visual Studio 2005 or one of the express editions.

1.4 Installation

Smart PropertyGrid.Net is supplied as a simple installer package. Simply run the executable and choose an installation directory. Once installed, an entry appears in the Start menu, letting you access files and options.

Includes

☑ Full featured sample application

☑ Quick Start guide / Associated sample

☑ Smaller sample application demonstrating TAB navigation

Evaluation copy

The evaluation library is obfuscated and displays a cross on the control as well as a copyright notice when run. The evaluation period and corresponding support last 30 days, which gives you plenty of time to see if it fits your needs.

To integrate the library with your own software, locate the license file (in the bin folder), and place it in the same folder as your executable or add an app.config file to your solution. This file will look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <appSettings>
        <add key="Xheo.Licensing.LicenseFolder" value="path_to_the_license_file" />
 </appSettings>
</configuration>
```

1.5 Redistribution

The SmartPropertyGrid.dll assembly must only be distributed with your application in an obfuscated form.

If you purchased the source code, a pre-built obfuscated assembly can be found in the “distrib” folder, otherwise in the “bin” folder.

When obfuscating your own software, embed the SPG assembly in your output executable. (Most obfuscators have this option.) If you purchased the source code, you can embed the assembly located in the “distrib” folder, or recompile the VisualHint namespace into your application.

If you have any difficulties or questions during this process, please contact the helpdesk.

1.6 Conventions used in this manual

This manual follows a few conventions:

"Property"/"property"	The words *property* and *properties* are be used in two ways:
"Properties"/"properties"	Upper case "P": Physical row in the PropertyGrid (This is the same convention as used in writing for the Property class) Lower case "p": C# property with its get and set accessor.
Italics	Indicates variables, classes, methods and properties inside the body of the text.
SPG	Smart PropertyGrid.Net
MSPG	Microsoft PropertyGrid
Code	When a method or property is called without any variable in front, the callee is assumed to be an instance of the PropertyGrid class.

2 PropertyGrid Content

A PropertyGrid is a two-column grid composed of Properties. Those Properties are organized under categories and sub-categories, like a conventional tree's nodes.

In SPG, a row, or a Property, corresponds to a .Net property, the identifier that you write containing a get and/or a set accessors.

However, in the SPG framework, a row or Property points to one of two kinds of data:

- Hosted by one of your classes
- Hosted internally by SPG

For example, you can specify a new row containing a boolean with an initial value of true, without having to bind to an actual property.

Accessing a Property

Whatever data it contains, you can access any Property or category by two means:

- A Property identifier
- A Property enumerator

2.1 Property identifiers

A Property identifier is a simple integer value attached to the Property. It is set on creation of the Property, but can be subsequently changed. You will usually set it as a unique identifier, but this is not mandatory.

Use the identifier to look for Properties:

```
mygrid.FindProperty(555);
```

(555 is the identifier of the lost Property.)

The identifier is also passed to all the callback methods. You can trigger actions according to its value by comparing it to known IDs.

2.2 Property enumerators

If you have a *PropertyEnumerator*, you can access the underlying Property:

```
Property prop = propEnum.Property;
```

A *PropertyEnumerator* is returned to you:

- When you create a new property.
- When you find a property.
- By an event (the enumerator is given to you in the argument of the event handler, or of the virtual method).

You can then use it as a parameter of other methods like:

```
myGrid.EnableProperty(propEnum, false);
```

You can also use it to navigate through the hierarchy of Properties.

The basic design under an enumerator can be represented by this simple diagram:

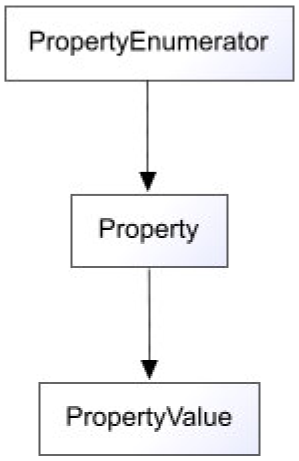


Figure 1 – Simple class diagram around the Property class

It is possible to store an enumerator for later use. However, A *PropertyEnumerator* is only valid in the view from which it originates. If the user switches, e.g., to the alphabetically sorted view, the *PropertyEnumerator* points to the right Property, but is no longer useful for navigating Properties.

2.3 Population modes

The PropertyGrid has three modes for populating the grid:

- Full Reflection Mode
- Dynamic Reflection Mode
- Full Dynamic Mode

2.3.1 Full Reflection Mode

Full Reflection Mode is familiar from Microsoft PropertyGrid. The content is set by using one of the following calls:

```
myGrid.SelectedObject = myTargetInstance;

myGrid.SelectedObjects = new object[] { myTargetInstance1, myTargetInstance2 };
```

All the target instance's public properties (with *BrowsableAttribute* set to true) are rendered in a two-level tree of root categories and Properties.

Notes:

When a target instance is already selected and you want to select a new one whose type is the same or similar, you may wish to save the states of the expanded Properties and what Property is selected. The following code will accomplish this task:

```
myGrid.BeginUpdate();
Hashtable states = myGrid.SavePropertiesStates();
myGrid.SelectedObject = myNewTargetInstance;
myGrid.RestorePropertiesStates(states);
myGrid.EndUpdate();
```

This is exactly what the *RefreshProperties* method does (it was called *Refresh* in the Microsoft PropertyGrid) except that it selects again the currently selected target instances. Calling this method is useful to reconstruct the content of the grid when one or some of the properties have been changed externally and a *TypeConverter* or a *TypeDescriptor* have published a new set of properties. Note that your stored *PropertyEnumerators* would become invalid.

```
myGrid.SelectedObject = myTargetInstance;
myTargetInstance.Font = null;
myGrid.RefreshProperties();
```

Customizing the grid content

Several techniques customize the grid content. Some are already available in the .Net framework (the classes *TypeConverter*, *ICustomTypeDescriptor* and *TypeDescriptionProvider* should be familiar). Others are introduced by SPG.

In this mode, one call to *SelectedObject(s)* fills the grid, so at this stage SPG cannot customize each Property. Instead, the .Net and the SPG framework use the attributes of each Property to alter their behavior or appearance.

Attributes from the .Net framework:

BrowsableAttribute	Filters out decorated properties.
DescriptionAttribute	Sets the comments text (at the bottom of the grid). To make this area visible: `CommentsVisibility = true;`
DisplayName	Sets the Property label.
MergablePropertyAttribute	Set to false to filter out decorated properties when *SelectedObjects* (note the final “s”) is used.

Attributes from the SPG framework:

PropertyIdAttribute

By default, the Categories have a negative identifier decrementing from -1 and Properties a positive identifier incrementing from 1. Use *PropertyIdAttribute* to assign a specific identifier:

```
[PropertyId(100)]
public int MyInteger {
    ...
```

SortedCategoryAttribute

Used during the initial sorting process – see “Sorting the Properties” on page 64.

SortedPropertyAttribute

Used during the initial sorting process- see “Sorting the Properties” on page 64.

ValueAddedToPropertyAttribute

This attribute defines a logical link between several data in the client application. This supports a single inplace control enabling the user to edit multiple values contained in a same target instance. For example, the “Unit” inplace control comprises: a textbox for editing a number and a combo box for selecting the units, e.g. currency.

```
[ValueAddedToProperty(PropertyUnitLook.UnitValue, "Frequency")]
public Units Unit { set; get; }

[PropertyLook(typeof(PropertyUnitLook))]
[PropertyFeel(PropertyGrid.FeelEditUnit)]
public int Frequency {
    ...
```

PropertyDisableAttribute

Used with no arguments, this sets a disabled Property:

```
[PropertyDisable]
public int MyProperty {
    ...
```

Used *with* arguments, it disables child Properties of a complex property. For example, this disables the Width property of a Pen:

```
[PropertyDisableAttribute("Width")]
public Pen MyPen {
    ...
```

PropertyHeightMultiplierAttribute

Sets the number of lines occupied by a Property. This works for inplace controls with multiple lines, e.g. multiline textboxes:

```
[PropertyHeightMultiplier(3)]
[PropertyFeel(PropertyGrid.FeelMultilineEdit)]
public string Answer {
    ...
```

Note that Properties displaying a set of possible values, such as checkboxes or radiobuttons, don’t need this attribute because the number of lines is automatically set.

PropertyManuallyDisabledAttribute

Displays a check box to the left of Property label, so that users can toggle between "enabled" and "disabled".

A typical use might be an optional command line parameter. For example, -p is optional, and can also have a value like -p4:

```
[DisplayName("-p")]
[PropertyManuallyDisabled]
public int paramP {
    ...
```

PropertyDropDownContentAttribute

Displays a custom editor inside the dropdown part of a combobox. For example, this displays a custom alpha color editor:

```
[PropertyDropDownContent(typeof(AlphaColorPicker))]
public Color MyColor {
    ...
```

This also works for a child Property whose code is inaccessible. The following assigns the same editor to the Color property of a Pen:

```
[PropertyDropDownContent("Color", typeof(AlphaColorPicker))]
public Pen MyPen {
    ...
```

See "FeelList and FeelEditList" on page 45.

PropertyFeel

Specifies the inplace control for a particular Property. A whole chapter is devoted to Feels – see page 37 - so here is a simple example which links multiple checkboxes to the *ParticipatingCountries* property:

```
[PropertyFeel(PropertyGrid.FeelCheckbox)]
public Countries ParticipatingCountries {
    ...
```

This also works for a child Property whose code is inaccessible. For example, this sets an up/down button Feel for a pen's Width property:

```
[PropertyFeel("Width", PropertyGrid.FeelEditUpDown)]
public Pen MyPen {
   ...
```

PropertyHide

Hides a Property. The Property is created, but remains invisible until shown.

For example, this hides a Property unneeded on first launch of an evaluation copy (because the user has yet to be invited to purchase the software):

```
[PropertyHide]
public string RegistrationKey {
    ...
```

This also works for a child Property whose code is inaccessible. The following hides the unfriendly CompoundArray property of a pen:

```
[PropertyHide("CompoundArray")]
public Pen MyPen {
    ...
```

PropertyLook

Specifies the kind of custom drawing for a particular Property value. The following sets a custom Look for a pen:

```
[PropertyLook(typeof(PropertyPenLook))]
public Pen MyPen {
    ...
```

This also works for a child Property whose code is inaccessible. For example, this draws the pen's Color property with a Look displaying its alpha component:

```
[PropertyLook("Color", typeof(PropertyAlphaColorLook))]
public Pen MyPen {
    ...
```

PropertyValidator

Assigns an object derived from *PropertyValidatorBase* to a Property so that its value is checked each time the user inputs new data. Here is an example:

```
[PropertyValidator(typeof(PropertyValidatorMinMax), 0, 100)]
public int Percentage {
    ...
```

The first parameter specifies the type of the validator class. The subsequent parameters are passed to the validator class constructor. There can be any number of these.

This also works for a child Property whose code is inaccessible. The following applies a validator to a Size property's Width and Height properties:

```
[PropertyValidator("Width", typeof(PropertyValidatorMinMax), 0, 20)]
[PropertyValidator("Height", typeof(PropertyValidatorMinMax), 0, 30)]
public Size MySize {
    ...
```

PropertyValueDisplayedAs

Specifies permitted values for a Property, typically one with a set of exclusive values edited using listboxes and radiobuttons. This enables you to avoid writing additional classes just for this one action.

This example modifies the strings for a Boolean to Yes and No:

```
[PropertyValueDisplayedAs(new string[] { "Yes", "No" })]
public bool MyBoolean {
    ...
```

This also works for a child Property whose code is inaccessible. Suppose a Font property is to have Booleans as checkboxes and display no strings:

```
[PropertyValueDisplayedAsAttribute("Bold", new string[2] { "", "" })]
[PropertyValueDisplayedAsAttribute("Italic", new string[2] { "", "" })]
[PropertyValueDisplayedAsAttribute("Strikeout", new string[2] { "", "" })]
[PropertyValueDisplayedAsAttribute("Underline", new string[2] { "", "" })]
public Font MyFont {
    ...
```

ShowChildProperties

This determines whether or not a Property can be expanded to make child Properties visible.

Take the example of our pen. In MSPG, no Properties would be visible without attaching a *TypeConverter* that acts like *ExpandableObjectConverter*. However, in SPG only the following is required:

```
[ShowChildProperties(true)]
public Pen MyPen {
    ...
```

It's just as simple to make child Properties unviewable. For example, the Font class has a type converter which publishes child Properties so that the Font node can be expanded. We can prevent this:

```
[ShowChildProperties(false)]
public Font MyFont {
    ...
```

! Unlike the *PropertyHide* attribute, this attribute prevents the creation of child Properties. You can't display them later without recreating the Property in the grid.

2.3.2 Dynamic Reflection Mode

Dynamic Reflection Mode gives full control of the dynamic content of new and existing grids. You can even use it to finish off the content drawn by "Full Reflection Mode" (see page 5), which actually uses Dynamic Reflection Mode internally.

This mode enables you to place Properties anywhere in the grid. The following code examples use Append operations, only. However, Insert operations also work.

Root categories

A root category must exist in order to host other Properties:

```
PropertyEnumerator rootEnum = AppendRootCategory(id, "Label");
```

The returned enumerator is used to create child Properties.

Subcategories

Not available when Full Reflection Mode is used.

Organize complex content by creating subcategories for root categories, or for other subcategories.

```
PropertyEnumerator subEnum = AppendSubCategory(parentEnum, id, "label");
```

The returned enumerator is used to create child Properties.

Properties

The following method appends a Property to the last position under a parent node:

```
public PropertyEnumerator AppendProperty(
    PropertyEnumerator underCategory,
    int id,
    string propName,
    object container,
    string memberName,
    string comment,
    params Attribute[] attributes)
```

Remarks

container Specifies the property's target instance.

You can combine different properties coming from different target instances.

memberName Specifies the Property display name.

attributes Any number of attributes for decorating the Property – see "

Attributes from the SPG framework" on page 6.

Some attributes do not work in this mode, e.g. *PropertyId* since the identifier is passed on the command line, and *SortedProperty* since calling *AppendProperty* controls the order of Properties at will.

Controlling Properties directly

In this mode, use some of the framework methods to directly control the effect of attributes. For example, the attribute *PropertyDisableAttribute* can be replaced by:

```
EnableProperty(propEnum, false);
```

2.3.3 Full Dynamic Mode

In Full Dynamic Mode, the framework completely hosts the Properties and their values. Categories and subcategories are created as in Dynamic Reflection Mode (see above page, 10). However, Property type and initial value are required, only. This mode uses neither target instance nor property name.

This is a convenient mode for filling a PropertyGrid instead of creating a specific class for storing highly dynamic data.

Example

A third party supplies code for an application's settings. A global shortcut is missing, but is required in the grid. You add it manually using the code:

```
propEnum = AppendManagedProperty(categoryEnum, id, "Global shortcut",
    typeof(Keys), Keys.F4, "");
```

Note The term *Managed Property* is distinct from *managed code* in .Net and simply reflects that SPG will manage the underlying data.

2.4 More on modes

2.4.1 Mixing the construction modes

It is possible to mix the three construction modes (see "Population modes" on page 5) when filling the grid, by changing Modes between Properties. For example, you could fill the grid with *SelectedObject(s)*, use Dynamic Reflection Mode to add Properties by targeting actual Properties, and finally use Full Dynamic Mode for additional “virtual” Properties.

Example

This shows existing properties created using Dynamic Reflection Mode and uses Full Dynamic Mode to add a virtual Property for quick information:

```
PropertyEnumerator propEnum = AppendProperty(categoryEnum, 1, "Width", target,
    "Width", "");

propEnum = AppendProperty(categoryEnum, 2, "Height", target, "Height", "");

propEnum = AppendManagedProperty(categoryEnum, 3, "Diagonal", typeof(double),
    0.0, "");
EnableProperty(propEnum, false);
```

This enables the end-user to quickly see the value of the diagonal formed by the width and the height of, e.g., a rectangle.

Of course, it's up to the developer to set the right value in the Diagonal Property (see "Handling the PropertyValue’s value" on page 51).

2.4.2 Hyperlinked Properties

If you already know the verbs in the Microsoft PropertyGrid, SPG hyperlinked Properties will seem familiar. SPG has two kinds of hyperlinks.

Clickable static text

The first kind of hyperlink is like a verb (term introduced by Microsoft in its PropertyGrid) embedded in the grid and occupies a full row in the middle of other Properties. It has no value and is just clickable static text.

To create this kind of hyperlink, use the following code:

```
propEnum = AppendHyperLinkProperty(parentEnum, id, "Click me", "Comment");
```

Hyperlinked property value

The second kind of hyperlink tags the value of a Property as if it were a web hyperlink, with the value displayed underlined in blue. The user CTRL + clicks the link to activate it.

You can also attach a link format to the value. For example, this returns a mail directive for a Property created to receive an email address:

```
propEnum = AppendManagedProperty(parentEnum, id, "Email", typeof(string),
    "webmaster@visualhint.com", "");
propEnum.Property.HyperLinkFormat = "mailto:{0}";
```

2.4.3 Multiple values per Property

You can store multiple data items in a single Property. The additional values can be found by reflection or created on the fly.

This feature is primarily used by the Unit look and feel classes displaying an editable numerical value and unit on the same row. However, it can be used in other ways.

Add a new value to an existing Property as follows:

```
Property.AddValue(object key, object container, string memberName,
    Attribute[] attributes);
```

Or:

```
Property.AddManagedValue(object key, Type valueType, object defaultValue,
    Attribute[] attributes);
```

Most of the arguments are the same as for calls to *AppendProperty* and *AppendManagedProperty*.

Notes

key	An identifier giving meaning to the additional value. E.g. if you create an inplace control making editable the three coordinates of a point in space, the keys must discriminate between X, Y and Z. This identifier is used by the attached look, feel, and inplace control classes. For an example, see "PropertyUnitLook" on page 35.

2.4.4 Multiple target instances per Property

SPG supports the *SelectedObjects* property familiar from MSPG. This produces Properties each pointing to the same set of target instances.

When not using the Full Reflection Mode, you can manually add a target instance to a Property:

```
PropertyEnumerator propEnum = AppendProperty(...);
AddTargetInstance(propEnum, targetInstance2, "PropertyName");
```

This creates a Property, and then links it to one called **PropertyName** in an object instance **targetInstance2**.

2.4.5 Hierarchy of Properties

A Property may often become a parent for other Properties:

- In the Full Reflection Mode, e.g. when a *TypeConverter* publishes properties (this is the case for a Font).
- In Dynamic Reflection and Full Dynamic Mode, when you use one of three manual options.

Automatic hierarchy

In Dynamic Reflection and Full Dynamic Mode, creating a single Property enables the framework to discover any child Properties at runtime:

```
PropertyEnumerator fontEnum = AppendProperty(rootEnum, 1, "Font", target, "Font", "");
```

Hierarchy with parent-child value dependencies

In Dynamic Reflection and Full Dynamic Mode, it is possible to create a hierarchy in which the value of the parent Property depends on those of the children.

Example

This example (from the sample) involves a target class for which the code is not available. The class comprises two integer Properties: “Width” and “Height”. However, you want to present these concatenated (e.g. "100; 200"), as the editable value of a single “Size” Property. This Property also expands to show the integers as the editable values of two child Properties. The user has a choice between editing the parent or the children.

```
PropertyEnumerator sizePropEnum = AppendManagedProperty(rootEnum, 1,
    "WindowSize", typeof(String), "", "");

PropertyEnumerator propEnum = AppendProperty(sizePropEnum, 2, "Width",
    target, "Width", "");

propEnum = AppendProperty(sizePropEnum, 3, "Height", target, "Height", "");
```

Use *PropertyValue.GroupedValueSeparator* to specify a separator other than “;”.

Hierarchy with no dependencies

In Dynamic Reflection and Full Dynamic Mode, use *NoLinkWithChildren* to create a hierarchy in which there are no dependencies between the values of parent and child.

Example

In a typical task management application, we would have: a task whose value is its name, plus sub Properties for the start and end dates:

```
PropertyEnumerator propEnum = AppendManagedProperty(rootEnum, 1, "Task",
typeof(String), "Work on 2.0", "");
propEnum.Property.Value.NoLinkWithChildren = true;

PropertyEnumerator propEnum2 = AppendManagedProperty(propEnum, 2, "Start date",
typeof(DateTime), new DateTime(2006, 7, 1), "");

propEnum2 = AppendManagedProperty(propEnum, 3, "End date", typeof(DateTime), new
DateTime(2006, 8, 31), "");
```

NoLinkWithChildren preserves the visual hierarchy between parent and children, but removes any dependency between their values.

2.4.6 Property Tags

When creating a Property at runtime, you can attach a Tag to it, or to its value. A tag can be any *ValueType*, or a reference to an object of known type:

```
propEnum.Property.Tag = 1000;
propEnum.Property.Value.Tag = "remind me later";
```

2.5 Browse the content

Once content is in place, you can use enumerators to navigate your Properties. Four classes, derived from *PropertyEnumerator*, are available.

2.5.1 Classes of enumerator

PropertyDeepEnumerator	A bidirectional enumerator, this browses all Properties, including the hidden ones, in display order.
PropertySiblingEnumerator	A bidirectional enumerator, this browses a parent's direct children, including the hidden ones. It only browses one level.
PropertyVisibleDeepEnumerator	As *PropertyDeepEnumerator*, except ignores hidden Properties.
PropertyVisibleSiblingEnumerator	As *PropertySibilingEnumerator*, except ignores hidden Properties.

Hidden Properties are those explicitly hidden by *ShowProperty*, not temporarily invisible because of a collapsed parent.

2.5.2 Returning an enumerator for browsing

Here are some different ways to get an enumerator to start browsing Properties:

Enumerator Class	Target Property	Action
Deep	A new Property.	Store the enumerator returned on creation (e.g. using *AppendProperty* or *AppendManagedProperty*).
Deep	A Property with a known identifier	*PropertyGrid.FindProperty(…)* (It is possible to search from a particular starting Property)
Deep	First Property in grid.	*PropertyGrid.FirstProperty*
Deep	Last Property in grid.	*PropertyGrid.LastProperty*
Visible Deep	First visible Property in grid.	*PropertyGrid.FirstVisibleProperty*
Deep	Parent of a Property with known enumerator	*PropertyEnumerator.Parent*
Sibling	Child of a Property with known enumerator	*PropertyEnumerator.Children*

2.5.3 Enumerator properties and methods

Useful *PropertyEnumerator* properties and methods include:

Action	*Properties and methods*
Return a count of a Property's direct children	*PropertyEnumerator.Count*
Return the collection limits	(Mainly used in loops. Also *RightBound* is returned by *FindProperty* when no property can be found) *PropertyEnumerator.LeftBound* *PropertyEnumerator.RightBound* **Note:** These enumerators don't point to valid Properties.
Clone an enumerator	Depending on the desired class: *GetDeepEnumerator()* *GetVisibleDeepEnumerator()* *GetSiblingEnumerator()*

	GetVisibleSiblingEnumerator()
	Clone()

Return whether or not the Property is at the root level	*PropertyEnumerator.HasParent*
Move an enumerator to another Property	*MoveFirst(), MoveNext()*, etc
Return the enumerator's Property instance	*PropertyEnumerator.Property*
Limit the enumerator to the current level and below	*PropertyEnumerator.RestrictedToThisLevelAndUnder*

Examples

Browse child Properties of a Property

```
PropertyEnumerator childrenEnumerator = propEnum.Children;
while (childrenEnumerator != childrenEnumerator.RightBound)
{
    // Do something here

    childrenEnumerator.MoveNext();
}
```

Browse all visible Properties in reverse order

```
PropertyEnumerator propEnum = FirstDisplayedProperty.MoveLast();
while (propEnum != propEnum.LeftBound)
{
    // Do something here

    vdEnumerator.MovePrev();
}
```

Browse from a Property to all its ancestors

```
PropertyEnumerator currentEnum = propEnum.Parent;
while (currentEnum != currentEnum.RightBound)
{
    // Do something here

    currentEnum = currentEnum.Parent;
}
```

3 Object Design

This class diagram shows the main relationships between the most commonly used classes:

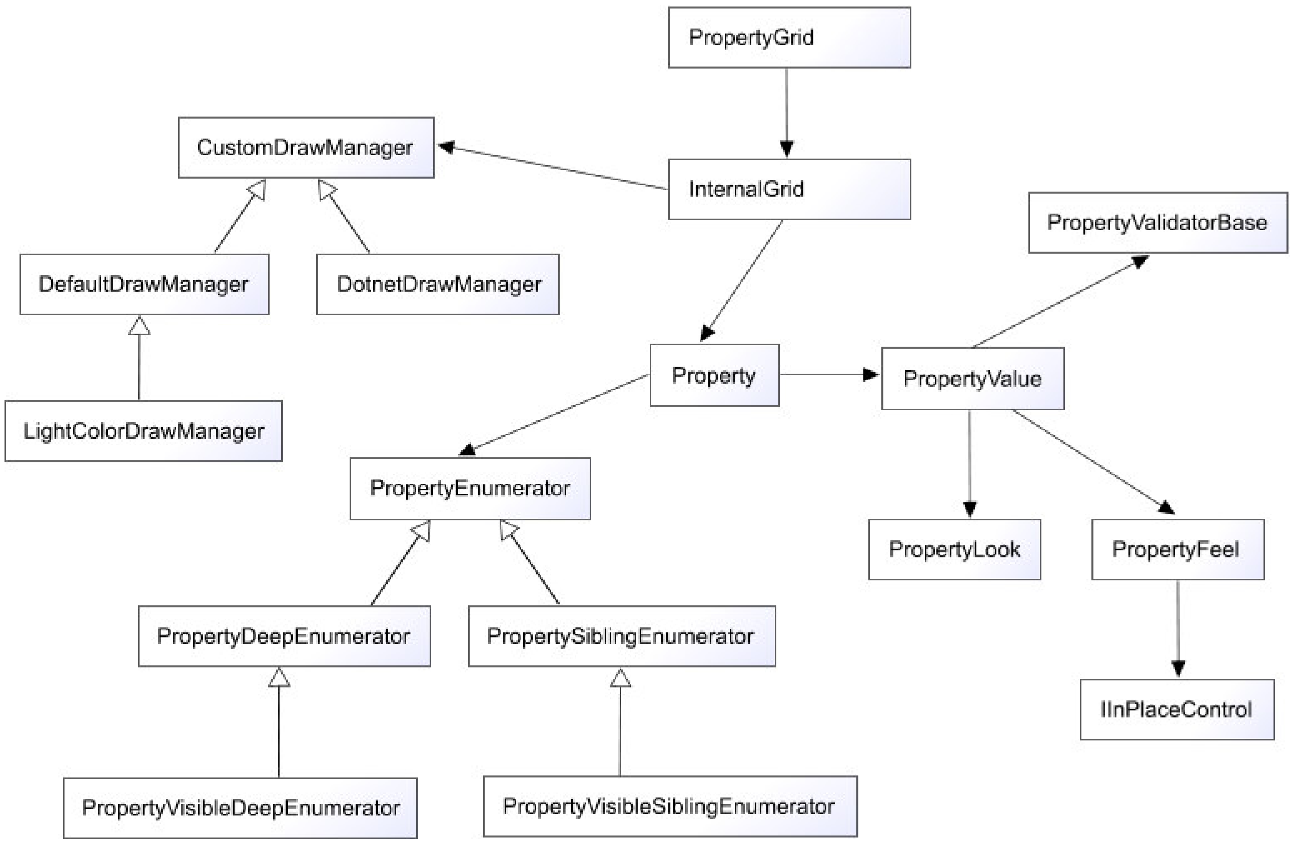


Figure 2 - General class diagram

PropertyGrid	Class dragged onto a form to instantiate a *PropertyGrid*. It contains an optional toolbar, the actual grid, and an optional comment area.
InternalGrid	Internal control handling the true two-columns grid. You never directly interact with this class. It contains a collection of Properties.
Property	One row in the grid. You usually use an enumerator to access this. It keeps a reference on a value.
PropertyValue	Class handling the value of the .Net property. Several internal types of values exist. These handle:

- Properties represented by *PropertyDescriptor* classes (found by reflection using *SelectedObject(s)* or *AppendProperty*)
- Properties simulated by a *TypeConverter*
- Virtual Properties (added using *AppendManagedProperty*).

PropertyLook	Handles custom painting for a Property.
PropertyFeel	Each kind of *PropertyFeel* (derived class) is a singleton that handles one kind of particular inplace control.
IInPlaceControl	Interface used by each kind of inplace control, which is above all a derived class of the WinForms *Control* class.
PropertyValidatorBase	Base class used by all validator classes, i.e. classes checking a user-entered value is correct.
PropertyEnumerator	Class that encapsulates a reference to a *Property* instance. An enumerator can be stored and used to navigate the collection of Properties. ▪ **PropertyDeepEnumerator**: A bidirectional enumerator, this browses all Properties, including the hidden ones, in display order. ▪ **PropertySiblingEnumerator**: A bidirectional enumerator, this browses a parent's direct children, including the hidden ones. It only browses one level. ▪ **PropertyVisibleDeepEnumerator**: As *PropertyDeepEnumerator*, except ignores hidden Properties. ▪ **PropertyVisibleSiblingEnumerator**: As *PropertySibilingEnumerator*, except ignores hidden Properties.
CustomDrawManager	Base class used to modify the global appearance of the PropertyGrid.

4 Events and Virtual Methods

The framework provides several events and virtual methods. These notify listeners or your class derived from *PropertyGrid* of various important events occurring in the system.

When an event is related to a particular Property (which is generally the case), the argument of the event contains a reference to an enumerator.

4.1 SelectedObjectChanged

- Triggered after SelectedObject(s) has been called.

4.1.1 Virtual method

```
protected override void OnSelectedObjectChanged(EventArgs e)
{
    // Add your code here...
    base.OnSelectedObjectChanged(e);
}
```

4.2 PropertyCreated

- Triggered on creation of a Property.
- Triggered for child Properties, then parent Properties.

4.2.1 Virtual method

This gives you the opportunity to further set the behaviour and appearance of a Property and its descendents:

```
protected override void OnPropertyCreated(PropertyCreatedEventArgs e)
{
    // Add your code here...
    base.OnPropertyCreated(e);
}
```

Example

The SPG framework does not have yet an attribute to set the description of Properties returned by a *TypeConverter*. Suppose that you create one derived from *DescriptionAttribute* and call it *DescriptionForChildAttribute* (the code for this is not shown here). You intend to use this attribute on complex properties like this:

```
[DescriptionForChild("GdiVerticalFont", "Indicates whether this Font is derived from a
GDI vertical font")]
public Font MyFont
{
  ...
```

You then decide to modify the description of any Property whose parent has the new attribute. The solution is to write a generic piece of code that will work for every Property that you create:

```
protected override void OnPropertyCreated(PropertyCreatedEventArgs e)
{
    Property parentProperty = e.PropertyEnum.Parent.Property;
    if ((parentProperty != null) && (parentProperty.Value != null))
    {
        List<Attribute> attrs = parentProperty.Value.
            GetAttributes(typeof(DescriptionForChildAttribute));
        foreach (DescriptionForChildAttribute attr in attrs)
        {
            Property property = e.PropertyEnum.Property;
            if (attr.ChildPropertyName == property.Name)
            {
                property.Comment = attr.Description;
                break;
            }
        }
    }
    base.OnPropertyCreated(e);
}
```

This code…

1) Checks a new Property's parent to see if it contains the new attribute.
2) If successful, loops through the attributes (because there can be multiple attributes for several child Properties).
3) If an attribute corresponds to the name of the current processed Property, applies the new comment (called Description in the MSPG).

4.3 PropertyPreFilterOut

By default, the framework filters out those properties with an attached BrowsableAttribute set to false. This event allows the client application to use its own criteria to filter the properties before they are created.

- Triggered for the Properties created when using *SelectedObject(s)*, only.
- Sub Properties returned by a TypeConverter are ignored.

4.3.1 Virtual method

```
protected override void OnPropertyPreFilterOut(PropertyPreFilterOutEventArgs e)
{
    // Add your code here...
    base.OnPropertyPreFilterOut(e);
}
```

The argument gives access to the new property's PropertyDescriptor. It also contains a property called FilterOut, which the client application can use to specify whether or not SPG filters out the new Property. This can take one of the following possible values:

Value	Effect on property
FilterDefault	Filtered out only if it has an attached BrowsableAttribute set to false.
FilterOut	Filtered out.
DontFilter	Not filtered out, even if it has an attached BrowsableAttribute set to false.

4.4 PropertyPostFilterOut

Gives the client application the opportunity to filter out some Properties of the grid after they are created.

- Triggered for the Properties created when using *SelectedObject(s)* only.
- Sub Properties returned by a TypeConverter are ignored.

4.4.1 Virtual method

Use the *FilterOut* = true to delete the new Property.

```
protected override void OnPropertyPostFilterOut(PropertyPostFilterOutEventArgs e)
{
    // Add your code here...
    base.OnPropertyPostFilterOut(e);
}
```

To prevent a Property from being created in the first place, set *BrowsableAttribute* to false, or use the PropertyPreFilterOutEvent with your own criteria. The advantage of *PropertyPostFilterOutEvent* is that you can refer to the Property with an enumerator (since it has been already created) and then delete it based on criteria such as location in the hierarchy, value, name, etc.

4.5 PropertySelected

- Triggered on selection of a Property.

4.5.1 Virtual method

```
protected override void OnPropertySelected(PropertySelectedEventArgs e)
{
    // Add your code here...
    base.OnPropertySelected(e);
}
```

4.6 PropertyExpanded

- Triggered on expansion or collapse of a Property.

4.6.1 Virtual method

```
protected override void OnPropertyExpanded(PropertyExpandedEventArgs e)
{
    // Add your code here...
    base.OnPropertyExpanded(e);
}
```

4.7 PropertyEnabled

- Triggered on disabling or enabling a Property.

4.7.1 Virtual method

```
protected override void OnPropertyEnabled(PropertyEnabledEventArgs e)
{
    // Add your code here...
    base.OnPropertyEnabled(e);
}
```

4.8 PropertyChanged

- Triggered when the value of a Property in the grid changes.

This is triggered once the end-user has successfully used an inplace control to modify the underlying value of a Property.

This is one of the most important events in the SPG framework.

4.8.1 Virtual method

```
protected override void OnPropertyChanged(PropertyChangedEventArgs e)
{
    // Add your code here...
    base.OnPropertyChanged(e);
}
```

Notes

An important feature is the way this virtual method can act on other Properties in the grid. How you use it depends on your application. For example, you may want to disable a Property's children when its value is set to false.

Example

This disables all children of a Property when its boolean value is set to false:

```
protected override void OnPropertyChanged(PropertyChangedEventArgs e)
{
    if (e.PropertyEnum.Property.Id == 1000)
    {
        bool value = e.PropertyEnum.Property.Value.GetValue();

        PropertyEnumerator propEnum = e.PropertyEnum.Children;
        propEnum.RestrictedToThisLevelAndUnder = true;
        while (propEnum != propEnum.RightBound)
        {
            EnableProperty(propEnum, value);
            propEnum.MoveNext();
        }
    }

    base.OnPropertyChanged(e);
}
```

Useful PropertyChangedEventArgs arguments

SelectedIndex	Index of item selected from a list (if applicable).
AdditionalValueKey	Identifier of modified value if it is the additional, rather than primary value of the Property see "Multiple values per Property" on page 13.

Tips

Since this event gives you a Property enumerator, you can access the PropertyValue instance to get useful data. For example:

The new value	e.PropertyEnum.Property.Value.GetValue().
The previous value	e.PropertyEnum.Property.PreviousValue. Use *PropertyValue.PreviousValueValid* to find out whether the previous value was undefined. This is useful, e.g., when the Property points to multiple Properties in multiple target instances.
The PropertyDescriptor of the value (if any)	e.PropertyEnum.Property.Value.PropertyDescriptor.

4.9 ValueValidation

- Triggered when a user-entered string cannot validate, or when, after it did not validate, it subsequently validates.

For detailed discussion of the value validation mechanism, please see “Validating the values” on page 53.

4.9.1 Virtual method

```
protected override void OnValueValidation(ValueValidationEventArgs e)
```

```
{
    // Add your code here...
    base.OnValueValidation(e);
}
```

4.10 InPlaceCtrlVisible

- Triggered when an inplace control becomes visible.

An inplace control appears on selection of a Property with an editable value, or on selection of a Property where a read-only value is available for selection and copying.

This event allows you to customize the way an inplace control works. You may have to repeat the customization every time the control becomes visible, because inplace controls are singletons.

Inplace controls are singletons in the SPG framework. This avoids creating and keeping all the controls for the grid's Properties in memory.

4.10.1 Virtual method

```
protected override void OnInPlaceCtrlVisible(InPlaceCtrlVisibleEventArgs e)
{
    // Add your code here...
    base.OnInPlaceCtrlVisible(e);
}
```

Example

Because a single up/down inplace control instance edits two Property values, it must be configured each time it appears for one of the two Properties:

```
protected override void OnInPlaceCtrlVisible(InPlaceCtrlVisibleEventArgs e)
{
    if (e.PropertyEnum.Property.Id == 1000)
        ((PropInPlaceUpDown)e.InPlaceCtrl).RealtimeChange = true;
    else if (e.PropertyEnum.Property.Id == 1001)
        ((PropInPlaceUpDown)e.InPlaceCtrl).RealtimeChange = false;

    base.OnInPlaceCtrlVisible(e);
}
```

4.11 InPlaceCtrlHidden

- Triggered when an inplace control becomes hidden.

This is the simpler counterpart of *InPlaceCtrlVisibleEvent*, and requires no special actions.

4.11.1 Virtual method

```
protected override void OnInPlaceCtrlHidden(InPlaceCtrlHiddenEventArgs e)
{
    // Add your code here...

    base.OnInPlaceCtrlHidden(e);
}
```

4.12 InPlaceCtrlCreated

- Triggered when an inplace control is created or recalled, before becoming visible.

This event is offered for the same reasons as the InPlaceCtrlVisibleEvent. It intervenes before the inplace control is visible, enabling you to make any configuration changes to the control that would recreate its handle.

In the example in the main sample code, the event is handled so as to set up the AutoComplete feature (.Net 2.0 only) for a combobox. This also works for textboxes.

4.12.1 Virtual method

```
protected override void OnInPlaceCtrlCreated(InPlaceCtrlCreatedEventArgs e)
{
    // Add your code here...
    base.OnInPlaceCtrlCreated(e);
}
```

4.13 PropertyUpDown

- Triggered when the user clicks on the arrows of an inplace control, or hits the up and down keys.

This allows you to specify a new value for the control.

4.13.1 Virtual method

```
protected override void OnPropertyUpDown(PropertyUpDownEventArgs e)
{
    // Add your code here...
    base.OnPropertyUpDown(e);
}
```

The argument contains:

- *Value*: the current value (as a string) of the Property. Modify this to set a new value.
- *ButtonPressed*: indicates what arrow has been pressed.

Example

```
protected override void OnPropertyUpDown(PropertyUpDownEventArgs e)
{
    if (e.PropertyEnum.Property.Id == 1000)
    {
        e.Value = (Double.Parse(e.Value) +
            (e.ButtonPressed == PropertyUpDownEventArgs.UpDownButtons.Up ?
                0.05: -0.05)).ToString();
    }

    base.OnPropertyUpDown(e);
}
```

4.14 PropertyButtonClicked

- Triggered when the user clicks the button of a *FeelButton* or *FeelEditButton* inplace control.

4.14.1 Virtual method

The typical use is to display a custom modal form for entering a new Property value. On closing the form, you must tell the framework if the value has actually changed. Do this by setting *PropertyButtonClickedEventArgs*'s *PropertyChanged* to true. (Typically, the value changes when the user clicks **OK**.)

```
protected override void OnPropertyButtonClicked(PropertyButtonClickedEventArgs e)
{
    // Add your code here...
    base.OnPropertyButtonClicked(e);
}
```

4.15 DisplayedValuesNeeded

- Triggered on creation of a property with a browsable list of possible values, e.g. listbox, radiobuttons, up/down buttons and checkboxes.

This event enables you to modify the set of possible values displayed by an inplace control at runtime.

4.15.1 Virtual method

```
protected override void OnDisplayedValuesNeeded(DisplayedValuesNeededEventArgs e)
{
    // Add your code here...
    base.OnDisplayedValuesNeeded(e);
}
```

Example

The following modifies a boolean which has default true/false strings:

```
protected override void OnDisplayedValuesNeeded(DisplayedValuesNeededEventArgs e)
{
    if (e.PropertyEnum.Property.Id == 1000)
        e.DisplayedValues = new string[] { "Enabled", "Disabled" };

    base.OnDisplayedValuesNeeded(e);
}
```

4.15.2 Changing the displayed values at runtime

Sometimes it is necessary to change the possible values that an inplace control displays at runtime. To do this you need to reset the displayed values.

<u>To trigger DisplayedValuesNeededEvent</u>

*) Use PropertyValue.ResetDisplayedValues(bool triggerEvent)

Depending on what you set in triggerEvent, the event is triggered immediately or when the values are actually needed.

To change the displayed values without triggering *DisplayedValuesNeededEvent*
*) Use PropertyValue.ResetDisplayedValues(string[] displayedValues)

4.16 HyperLinkPropertyClicked

Triggered by the user clicking:

- A Property created by a call to *AppendHyperLinkProperty*
- A Property with a *HyperlinkFormat* set

4.16.1 Virtual method

```
protected override void OnHyperLinkPropertyClicked(
    PropertyHyperLinkClickedEventArgs e)
{
    // Add your code here...
    base.OnHyperLinkPropertyClicked(e);
}
```

Reading the modified hyperlink value

If the Property is a regular Property with a value, you can read the hyperlinked value modified by the string hyperlink format using *PropertyHyperLinkClickedEventArgs.HyperLink*. (For static hyperlinks, this property equals an empty string.)

4.17 DisplayModeChanged

- Triggered when the display mode (*Categorized*, *Flat*, *FlatSorted*) changes.

4.17.1 Virtual method

```
protected override void OnDisplayModeChanged(EventArgs e)
{
    // Add your code here...
    base.OnDisplayModeChanged(e);
}
```

4.18 DrawingManagerChanged

- Triggered when the drawing manager changes.

This is used whenever the new drawing manager must be configured.

4.18.1 Virtual method

```
protected override void OnDrawingManagerChanged(EventArgs e)
{
    // Add your code here...
    base.OnDrawingManagerChanged(e);
}
```

Example

This sets some properties of the *LightColorDrawManager* when it is applied to the grid:

```
protected override void OnDrawingManagerChanged(EventArgs e)
{
    if (DrawingManager == DrawManagers.LightColorDrawManager)
    {
        LightColorDrawManager manager = DrawManager as LightColorDrawManager;
        manager.SetCategoryBackgroundColors(
            Color.FromArgb(213, 224, 239), Color.White);
        manager.SetSubCategoryBackgroundColors(
            Color.FromArgb(213, 239, 224), Color.White);
    }

    base.OnDrawingManagerChanged(e);
}
```

5 Displaying a value

The value part of a Property can be drawn in two ways:

- Generically by the *PropertyValue* class drawing engine
- By an optional *PropertyLook* class attached to the Property

5.1 Generic display

In generic display, the drawing engine:

1) Determines if the Property has an attached *UITypeEditor*.
2) Determines if the value has an attached *ImageList.*
3) Uses a string to represent the value.

For a detailed explanation of each step, see below:

5.1.1 Determines if the Property has an attached UITypeEditor

The drawing engine first determines if the Property has an attached *UITypeEditor*.

Notes

- MSPG uses the *UITypeEditor* class, part of the .Net framework (see “Editing a value” on page 37) to provide an editing mechanism for a given data type.
- Some editors draw a small representation of the value at the left of the string, e.g. the *ColorEditor*. That’s why the SPG code checks to see if such an editor is available for the drawing and assigns it, unless told otherwise.

5.1.2 Determines if the value has an attached ImageList

The drawing engine next determines if the value has an attached *ImageList*, and if a current image index has been set.

To set the current image index

```
PropertyValue value = propEnum.Property.Value;
value.ImageList = ...;
value.CurrentImageListIndex = ...;
```

To control the size of the displayed image

```
value.ImageSize = new Size(16, 16);
```

If you don’t specify a size, the engine uses the Property height.

5.1.3 Uses a string to represent the value

Finally, the drawing engine uses a string to represent the value. This string is returned by the TypeConverter class attached to the property or property type.

If you specify your own TypeConverter classes, be aware that SPG supplies a context instance in its various Convert methods. This is a *PropertyTypeDescriptorContext* class, that:

- Correctly sets the *Instance* and the *PropertyDescriptor* properties
- Supplies a reference to a *PropertyEnumerator* object that points to the Property.

Generally speaking, each time an *ITypeDescriptorContext* class is expected, SPG supplies its own *PropertyTypeDescriptorContext*.

The *CultureInfo* parameter, which is customizable per Property, is also passed to TypeConverter classes.

5.2 Optional PropertyLook class

The *PropertyLook* class - and any custom class derived from it - displays values in ways not handled by the generic engine.

SPG comes with a number of built-in looks.

In most cases, when you assign a look to a Property, you must also assign the corresponding editing mechanism. For example, if you display an enumeration as radio buttons, the user does not expect to be presented with a generic listbox for editing!

5.2.1 Assigning a look

It is possible to assign a look with an attribute. This is useful for Properties created in Full Reflection Mode.

Example

This example makes full use of the *PropertyLook* attribute:

```
[PropertyLook(typeof(PropertyPenLook))]
[PropertyLook("Color", typeof(PropertyAlphaColorLook))]
public Pen OutlinePen {
    ...
```

- The first line instructs SPG that the *OutlinePen* property will be depicted with a *PropertyPenLook* class.
- The second line assigns a specific look class to the **Color** Property – a very powerful way to customize sub properties whose code is inaccessible.

Setting a look at runtime

It is also possible to set a look at runtime:

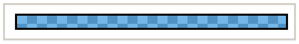
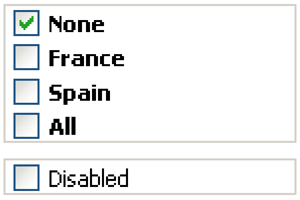
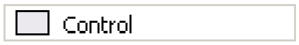
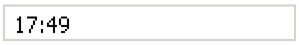
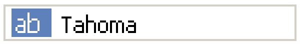
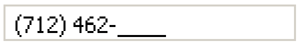
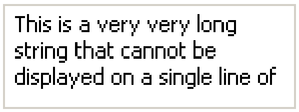
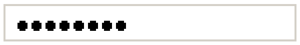
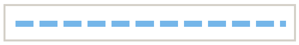
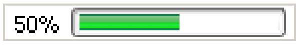
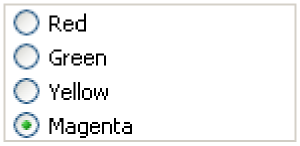
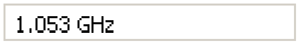
```
propEnum.Property.Value.Look = new PropertyProgressBarLook(true, "{0}%");
```

Configuring a look at runtime

Setup a look at runtime by casting it to the proper class. For the previous example, the code would be:

```
(propEnum.Property.Value.Look as PropertyProgressBarLook).Format = "{0}px";
```

5.2.2 SPG built-in Look classes

PropertyAlphaColorLook 	*PropertyCheckboxLook*  ✓ None, France, Spain, All; Disabled	*PropertyColorLook*  Control
PropertyDateTimeLook  17:49	*PropertyFontNameLook*  ab Tahoma	*PropertyMaskedEditLook*  (712) 462-____
PropertyMultilineEditLook  This is a very very long string that cannot be displayed on a single line of	*PropertyPasswordLook*  ●●●●●●●●	*PropertyPenLook* 
PropertyProgressBarLook  50%	*PropertyRadioButtonLook*  Red, Green, Yellow, ● Magenta	*PropertyUnitLook*  1.053 GHz

PropertyAlphaColorLook

- Draws a classic checkerboard behind the chosen transparent color.

PropertyCheckboxLook

- Draws one or more checkboxes.

It must be associated with boolean, enumeration or with ICollection data types.

PropertyColorLook

- Draws a simple color box with the color data just inside.

This is identical in function to the *ColorEditor* in the .Net framework, but offers improved drawing of the content.

PropertyDateTimeLook

- Displays a date/time string with the correct format (time, short date, long date), in the same manner as the true .Net *DateTimePicker* control.

The class has two properties which act like their *DateTimePicker* counterparts:

- Format
- CustomFormat

Set them on the look so that the inplace control knows how to configure itself:

```
propEnum = AppendProperty(upDownCategory, _id++, "Time", this, "Time", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelDateTime);
propEnum.Property.Look = new PropertyDateTimeLook("HH:mm");
```

PropertyFontNameLook

- Shows the name of a font with a small sample at the left side.

This is identical in function to the *FontNameEditor* in the .Net framework, but offers improved drawing of the content.

PropertyMaskedEditLook

- Draws a value exactly as drawn by a *MaskedTextBox*.

You can pass the *Mask* and the *PromptChar* in the constructor, or later, using properties with the same names.

PropertyMultilineEditLook

- Forces a multiline inplace control to display text as rendered in a multiline textbox.

Example

Here, the *HeightMultiplier* property sets the row height as a multiple of the single line height:

```
propEnum = AppendProperty(parentEnum, id, "Multiline", this, "MyString", "");
propEnum.Property.Value.Feel = GetRegisteredFeel(PropertyGrid.FeelMultilineEdit);
propEnum.Property.HeightMultiplier = 3;
propEnum.Property.Value.Look = new PropertyMultilineEditLook();
```

PropertyPasswordLook

- Displays a value as bullets.

You can apply this look explicitly, or set the Property's *PasswordPropertyText* attribute so that this look will be selected along with the correct inplace control.

PropertyPenLook

- Displays a line as if drawn by the Property’s underlying pen.

This provides an intuitive way of selecting a pen. A Property with this look cannot have an inplace control (i.e. no feel), since it makes no sense for the user to enter a string.

PropertyPenLook is the typical example of a look that exploits all the drawing area to represent the underlying data.

PropertyProgressBarLook

- Shows a progress bar plus optional non-editable value for an integer property.

A Property with this look cannot have an inplace control, since it makes no sense for the user to enter a string.

The value can be formatted:

```
propEnum = AppendManagedProperty(parentEnum, id, "Progress", typeof(int), 0, "");
propEnum.Property.Look = new PropertyProgressBarLook(true /* drawFrame*/, "{0}%");
```

Set the progress bar bounds by applying a *PropertyValidatorMinMax* validator:

```
propEnum.Property.Value.Validator = new PropertyValidatorMinMax(0, 100);
```

PropertyRadioButtonLook

- Draws multiple radiobuttons, one for each possible value.

This is generally used for enumerations, but could also work for any type with a set of possible values. (A simple string can use this look as long as you supply possible values).

Example

```
propEnum = AppendProperty(parentEnum, id, "Radio", this, "MyProperty", "");
propEnum.Property.Look = new PropertyRadioButtonLook();
```

It is not necessary to set the height multiplier. The framework deduces this from the possible displayed values attached to the Property.

PropertyUnitLook

- Displays a value and its unit as a single.

When the Property is selected, the *FeelEditUnit* inplace control shows a textbox for a numerical value and a combobox for its unit. When it is not selected, the value and its unit must be displayed as a single string, which is what this look does.

Example

```
propEnum = AppendProperty(parentCategory, id, "Frequency", this, "MyFreq", "");
propEnum.Property.AddValue(PropertyUnitLook.UnitValue, this, "MyUnit", null);
propEnum.Property.Look = new PropertyUnitLook();
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelEditUnit);
```

The look is assigned to the Property, but in order to work, must be aware of the Unit value. The *AddValue* call does this, and passes a predefined identifier *PropertyUnitLook.UnitValue* linking the Property to the **MyUnit** enumeration property.

5.2.3 Creating your own Look classes

When creating your own Look classes, you may find it helpful to base your code on SPG's existing *PropertyLook* classes.

You can register your new class with SPG so that it will be automatically selected for Properties of a certain type.

Example

```
RegisterLookAttachment(typeof(Pen), typeof(PropertyPenLook));
```

6 Editing a value

The SPG framework provides a large set of inplace controls. An inplace control is the component appearing in the right column when a Property containing a value is selected and not disabled.

Assign inplace controls using Property feels. A feel is actually an instance of a class derived from *PropertyFeel*. The feel creates and correctly positions the inplace control.

6.1 Automatic feel assignment

When a Property is first created, SPG assigns a feel according to a simple order of preference.

SPG order of preference for assigning a feel

1) Property attribute, *PropertyFeelAttribute*.
2) Parent Property attribute, *PropertyFeelAttribute* specifying feel of child.
3) Property attribute, *PasswordPropertyTextAttribute*.
4) Feel registered for the underlying Property type.
5) *UITypeEditor* for the underlying Property type.
6) Set of standard values published by the Property value's underlying *TypeConverter*.
7) Default feel.

For detailed description of each preference see below.

It is also possible to assign a feel at runtime. See "Manually assigning a feel" on page 40.

To assign a Property feel, we use an identifier string as defined in the *PropertyGrid* class:

Identifier	Description
FeelNone	No feel is used on the Property.
FeelEdit	A simple textbox.
FeelMaskedEdit	A masked textbox.
FeelUpDown	Up/Down buttons at the right side without textbox.
FeelEditUpDown	Up/Down buttons at the right side with a simple textbox.
FeelEditPassword	Textbox to edit a password (bullet characters).
FeelList	Dropdown list without textbox.
FeelEditList	Dropdown list with a textbox.
FeelButton	Button at the right side without textbox.
FeelEditButton	Button at the right side with a textbox.

Identifier	Description
FeelFontButton	Button at the right side with a textbox. Clicking on the button opens the usual font dialog editor.
FeelDateTime	True DateTimePicker control for dates and times.
FeelCheckbox	One or multiple checkboxes.
FeelRadioButton	Multiple radiobuttons.
FeelTrackbar	Trackbar showing the current value on the left.
FeelTrackbarEdit	Trackbar showing the current value on the left in a textbox.
FeelMultilineEdit	Multi-line textbox.

UITypeEditor classes

There is no identifier for *UITypeEditor* classes because it is impossible to target a specific editor from the range of possible editors. For this reason, the feel for *UITypeEditor* classes is handled internally.

6.1.1 PropertyFeelAttribute

Specify the feel for a Property at design-time or at runtime by passing the *PropertyFeelAttribute* class to methods like *AppendProperty*.

Example

```
[PropertyFeel(PropertyGrid.FeelEditList)]
public string OperatingSystems {
    ...
```

The Property is assigned a combobox.

This may seem strange for a string, but is possible in SPG. The listbox can be populated on demand by dynamic data, and the result of the selection is set in the string.

6.1.2 Parent's PropertyFeelAttribute

Specify the feel for a child Property at design-time or at runtime by passing the *PropertyFeelAttribute* class to methods like *AppendProperty*. This approach is useful in Full Reflection Mode and to explicitly decorate an inaccessible child Property.

Example

```
[PropertyFeel("MiterLimit", PropertyGrid.FeelEditUpDown)]
[PropertyFeel("Color",PropertyGrid.FeelList)]
[PropertyFeel(PropertyGrid.FeelNone)]
public Pen OutlinePen {
    ...
```

The Property is instructed to assign specific feels to two child Properties: *Color*, and *MiterLimit*. We don't have the code for the Pen class, so this is an easy way to specify their editor - more convenient than writing a new *TypeDescriptor* and assigning it to the pen.

The Property for the Pen itself has no feel, thanks to the *FeelNone* identifier. (Note that "null" could have been used instead.)

6.1.3 PasswordPropertyTextAttribute

The *FeelEditPassword* identifier assigns a password feel in the usual way. However, if you have an existing class that uses the *PasswordPropertyText* attribute, the correct feel will be chosen for you.

This attribute also assigns the correct look class.

6.1.4 Feel registered for the Property type

There are two kinds of registration:

Assigning a string identifier to a custom feel

A custom feel must be registered before you can use it in the grid. This involves using the *RegisterFeel* call to assign a string identifier, and then informing the SPG framework:

```
public const string MyFeelIdentifier = "myidentifier";
...
myGrid.RegisterFeel(MyFeelIdentifier, new MyCustomFeel(myGrid));
```

This creates the feel as a singleton and stores it in the PropertyGrid.

Associating a feel with a Property type

You can make a custom or built-in feel the default for new Properties of a specified type. (Different feels can subsequently be applied.) Specify a default association by registering a pair (Property type and feel identifier) with the *PropertyGrid* class, e.g.

```
RegisterFeelAttachment(typeof(Font), FeelFontButton);
```

6.1.5 UITypeEditor for the property type

If the SPG framework fails to find a feel specified for the property or its type (see above), it scans the underlying type of the Property for a *UITypeEditor* class that can handle the type. It looks for this class in the *EditorAttribute* placed in the metadata of the property type, then in custom attributes passed to an eventual *Append(Managed)Property* call. Even if the property type is a nullable type, the editor will be found.

If SPG finds a *UITypeEditor* class with no particular style (*GetEditStyle()* returns *UITypeEditorEditStyle.None*), it checks the *TypeConverter* to confirm that the editor can handle standard values (GetStandardValuesSupported() returns true). This is typically what happens, e.g., for the *FontNameEditor* class. This class offers a useful way to paint a font name (it draws sample text using the selected font) and also uses a generic list which handles all possible values. In such cases, the SPG framework returns a *FeelEditList* identifier.

6.1.6 Default feel for lists

On failing to satisfy the above preferences, SPG interrogates the *TypeConverter* to see if it supports standard values. If this is the case, it returns a *FeelEditList* identifier.

6.1.7 Global default feel

Finally, the framework returns the default feel. If none is specified, it returns the *FeelEdit* identifier.

To set the default feel
Use the *PropertyGrid.DefaultFeel* property.

6.2 Manually assigning a feel

You can change the feel of a Property at any time by directly setting the feel of the Property class with the one returned by *PropertyGrid.GetRegisteredFeel()*:

```
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelMaskedEdit);
```

It is possible to avoid inplace controls for a given Property:

```
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelNone);
// or
propEnum.Property.Feel = null;
```

Technically speaking, the feel belongs to a *PropertyValue* instance. But as shown, there is a shortcut to access it directly from the Property class. The following does the same as the above example, but only if the Value instance is non null (and this time we modify the default feel):

```
propEnum.Property.Value.DefaultFeel = null;
```

6.3 Feels in detail

This section covers feels in detail, including conditions for their use, data types, and how the client code is implemented depending on the mode used to create Properties.

6.3.1 FeelNone

- The equivalent of null.

This identifier exists for readability and can be replaced by null. Use it when you do not need an inplace control to appear. The user will be unable to select the text (if any) for copying.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelNone)]
public string MyProperty {
    ...
```

This is equivalent of:

```
[PropertyFeel(null)]
public string MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelNone);
```

This is equivalent of:

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = null;
```

6.3.2 FeelEdit

- Creates a textbox.

The textbox is the most simple inplace control. It does support clipboard operations. Text boxes displayed by other feels belong to the same textbox class as displayed by this class.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelEdit)]
public string MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelEdit);
```

6.3.3 FeelMaskedEdit

- Creates an inplace control derived from the *MaskedTextBox* class.

A *PropertyMaskedEditLook* must also be assigned. The Mask and the *PromptChar* properties are set on the look.

Configuring a Property

<u>At design-time</u>

```
[PropertyFeel(PropertyGrid.FeelMaskedEdit)]
[PropertyLook(typeof(PropertyMaskedEditLook), "(000) 000-0000"), '_']
public string Editbox11 {
    ...
```

<u>At runtime</u>

```
propEnum = AppendProperty(parentEnum, id, "Phone#", this, "Phone", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelMaskedEdit);
propEnum.Property.Look = new PropertyMaskedEditLook("(000) 000-0000", '_');
```

6.3.4 FeelEditPassword

- Configures a simple textbox so that the string is displayed with bullets.

It must be used in conjunction with the *PropertyPasswordLook* class.

Configuring a Property

<u>At design-time</u>

```
[PropertyFeel(PropertyGrid.FeelEditPassword)]
[PropertyLook(typeof(PropertyPasswordLook))]
public string Password {
    ...
```

As a shortcut, use the *PasswordPropertyText* attribute defined in the .Net framework:

```
[PasswordPropertyText(true)]
public string Password {
    ...
```

<u>At runtime</u>

```
propEnum = AppendProperty(parentEnum, id, "Password", this, "Password", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelMaskedEdit);
propEnum.Property.Look = new PropertyPasswordLook();
```

6.3.5 FeelMultilineEdit

- Turns a simple textbox into a multiline textbox.

The height is fixed and must be supplied as a multiplier of the basic row height.

This feel must be used in conjunction with the *PropertyMultilineEditLook* class.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelMultilineEdit)]
[PropertyHeightMultiplier(3)]
[PropertyLook(typeof(PropertyMultilineEditLook))]
public string Editbox3 {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelMultilineEdit);
propEnum.Property.HeightMultiplier = 3;
propEnum.Property.Look = new PropertyMultilineEditLook();
```

6.3.6 FeelUpDown and FeelEditUpDown

- Sets up an up/down button to the right of an optional textbox.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelUpDown)]
public string MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelUpDown);
```

Setup

At both design-time and runtime, you can extend the inplace control's possible behaviour by configuring it the moment it appears. Do this during the *InPlaceCtrlVisibleEvent* event by casting the inplace control to the correct type and calling some properties:

```
protected override void OnInPlaceCtrlVisible(InPlaceCtrlVisibleEventArgs e)
{
    if (e.PropertyEnum.Property.Id == myId)
        ((PropInPlaceUpDown)e.InPlaceCtrl).RealtimeChange = true;

    base.OnInPlaceCtrlVisible(e);
}
```

Accessible properties include:

Increment	Increment used when the property is a numerical value.
RealtimeChange	If true, the arrows and up/down keys immediately change the value. If false, the value is validated only on leaving the control.

Using a validator class to set limits

Validator classes can set limits on up/down buttons (see "Validating the values" on page 53). SPG comes with a min/max validator for this purpose. Assign it to the Property and the feel will detect it:

```
[PropertyValidator(typeof(PropertyValidatorMinMax), 0, 200)]
public int MyProperty {
...
```

Property types

The up/down inplace control can be used for:

☑ Any Property type that has a set of different possible values (like enumeration, boolean, color, etc).

☑ A numerical type. (As seen above you can modify the increment.)

☑ Any other type (like a string) as long as you catch the *PropertyUpDownEvent* and supply the next value. See the example, below.

Example (from the sample code)

```
protected override void OnPropertyUpDown(PropertyUpDownEventArgs e)
{
    if (e.PropertyEnum.Property.Id == 1000)
    {
        e.Value = (Double.Parse(e.Value) + (e.ButtonPressed ==
           PropertyUpDownEventArgs.UpDownButtons.Up ? 0.05 : -0.05)).ToString();
    }
    else if (e.PropertyEnum.Property.Id == 2000)
    {
        int index = upDownString.IndexOf(e.Value);
        try
        {
           e.Value = (string)upDownString[index + (e.ButtonPressed ==
                PropertyUpDownEventArgs.UpDownButtons.Up ? -1 : 1)];
        }
        catch (ArgumentOutOfRangeException)
        {
        }
    }
    base.OnPropertyUpDown(e);
}
```

The first *If* block is equivalent to setting an increment of 0.05. The second *If* block changes the value of a string property using dynamic values from the *upDownString* array.

SPG is very flexible. *PropertyValueDisplayedAsAttribute* class would also define a set of possible values for this string. You can even change the values at runtime using a call to *PropertyValue.ResetDisplayedValues()*, or by listening for the *DisplayedValuesNeededEvent* event (See “Events and Virtual Methods” on page 20).

6.3.7 FeelList and FeelEditList

- Creates an inplace control capable of opening a dropdown listbox and showing an optional textbox.

By default (for the enum, string and boolean types), the listbox is the classic list of strings showing all the possible values of the property.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelEditList)]
public string MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelEdiList);
```

Setup

At both design-time and runtime, you can extend the inplace control's possible behaviour by configuring it the moment it appears. Do this when handling the *InPlaceCtrlVisibleEvent* event, by casting the inplace control to the correct type and calling some properties:

```
protected override void OnInPlaceCtrlVisible(InPlaceCtrlVisibleEventArgs e)
{
    if (e.PropertyEnum.Property.Id == myId)
        ((PropInPlaceList)e.InPlaceCtrl).RealtimeChange = true;
    base.OnInPlaceCtrlVisible(e);
}
```

Accessible properties include:

RealtimeChange	If true, the up/down keys immediately change the value.
	If false, the value is validated only on leaving the control.

Property types

The listbox inplace control can be used for:

☑ Any Property type that has a set of different possible values (like enumeration, boolean, color, etc).

☑ Any other type (like a string) as long as you supply the values. See the example, below.

Example (from sample code)

```
protected override void OnDisplayedValuesNeeded(DisplayedValuesNeededEventArgs e)
{
    if (e.PropertyEnum.Property.Id == 1000)
    {
        e.DisplayedValues = new string[] { "Mercury", "Venus", "Earth", "Mars",
            "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto" };
    }
    base.OnDisplayedValuesNeeded(e);
}
```

Custom control for the listbox

It is possible to customize the content of the listbox using a control implementing the *IDropDownContent* interface.

Example

An example in the sample code edits an alpha color. This involves using the *PropertyDropDownContentAttribute* class on a property or as an argument of methods like *AppendProperty*. You can register your custom control so that it is automatically selected for a particular Property type:

```
RegisterDropDownContent(typeof(Enum), typeof(DropDownContentListBox));
```

6.3.8 FeelButton and FeelEditButton

- Creates an inplace control with a button to the right of the string value and an optional textbox.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelButton)]
public string MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid. FeelButton);
```

Setup

At both design-time and runtime, you can extend the button's possible appearance by configuring it the moment it appears. This is done when handling the *InPlaceCtrlVisibleEvent* event, by casting the inplace control to the correct type and calling some properties:

```
protected override void OnInPlaceCtrlVisible(InPlaceCtrlVisibleEventArgs e)
{
    if (e.PropertyEnum.Property.Id == myId)
        ((PropInPlaceButton)e.InPlaceCtrl).ButtonText = "Add...";

    base.OnInPlaceCtrlVisible(e);
}
```

Accessible properties include:

ButtonText	Sets the caption of the button.

6.3.9 FeelFontButton

- Creates an inplace control showing a button at the right of the string value plus an optional textbox.

When the button is clicked, a Font dialog editor opens.

Configuring a Property

At design-time

```
public Font MyFont {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Font", this, "MyFont", "");
```

No feel is explicitly attached. This is because the framework automatically registers some feels (and some looks) with types.

6.3.10 FeelDateTime

- Creates a *DateTimePicker* control, supporting editing of both date and time. It supports only an updown button, no dropdown month calendar.

The MSPG offers only a textbox equipped with a month calendar control to edit a *DateTime* property. SPG, however, supplies a true *DateTimePicker* control that enables editing a time or a date field by field with an updown button. When using this feel, you must also provide a *PropertyDateTimeLook* (see "SPG built-in Look classes" on page 33).

Configuring a Property

At design-time

```
[PropertyLook(typeof(PropertyDateTimeLook), DateTimePickerFormat.Short)]
[PropertyFeel(PropertyGrid.FeelDateTime)]
public DateTime DateTime
{
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Date", this, "MyDate", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelDateTime);
propEnum.Property.Look = new PropertyDateTimeLook(DateTimePickerFormat.Short);
```

Setup

You can set the usual properties for *DateTimePicker* controls using the look class. At runtime this is trivial:

```
(propEnum.Property.Look as PropertyDateTimeLook).CustomFormat = "HH:mm";
```

At design time, you need to explicitly assign the look:

```
[PropertyLook(typeof(PropertyDateTimeLook), "HH:mm")]
public DateTime DateTime {
    ...
```

Note:The DateTimePicker control from Microsoft does not accept null values. Therefore, it is impossible to use this feel when using a nullable *DateTime?*, or when using *SelectedObjects*.

6.3.11 FeelCheckbox

- Creates an inplace control showing one or more checkboxes.

This must be used in conjunction with the PropertyCheckboxLook class.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelCheckbox)]
[PropertyLook(typeof(PropertyCheckboxLook))]
public bool MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid. FeelCheckbox);
propEnum.Property.Look = new PropertyCheckboxLook();
```

Property types

The checkbox inplace control can be used for:

☑ Any Property type that has a set of different possible values (like enumeration, boolean, color, etc).

Only one checkbox is shown for booleans.
Enumerations must be marked with the [*Flags*] attribute.

☑ Any other type (like a string) as long as you supply the set of possible values, see the example below.

Example

This sets the possible value for two different properties of the same type at design-time:

```
[Flags]
public enum Countries
{
    NoCountry = 0,
    Country1 = 1,
    Country2 = 2,
    AllCountries = 3
}

private Countries _countries = Countries.NoCountry;

[PropertyValueDisplayedAs(new string[] { "None", "France", "Spain", "All" })]
public Countries Countries_set1 {
...
[PropertyValueDisplayedAs(new string[] { "None", "Germany", "Ireland", "All" })]
public Countries Countries_set2 {
...
```

Dynamic sets of values

This feel can also handle dynamic sets of values. Possible values must be supplied in a collection. Selected values must be stored in an *ArrayList*, since more than once item may be selected:

```
private ArrayList _projects = new ArrayList();
public ArrayList Projects {
    ...
List<string> stdValues = new List<string>();
stdValues.Add("Windows 2000");
stdValues.Add("Windows XP");
stdValues.Add("Windows Server 2003");
stdValues.Add("Windows Vista");

propEnum = AppendProperty(subEnum, _id++, "Projects", this, "Projects", "",
    new PropertyValueDisplayedAsAttribute(stdValues));
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelCheckbox);
propEnum.Property.Value.Look = new PropertyCheckboxLook();
```

6.3.12 FeelRadioButton

- Creates an inplace control showing multiple radiobuttons.

This feel must be used in conjunction with the PropertyRadioButtonLook class.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelRadioButton)]
[PropertyLook(typeof(PropertyRadioButtonLook))]
public string MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid. FeelRadioButton);
propEnum.Property.Look = new PropertyRadioButtonLook();
```

Property types

The radiobutton inplace control can be used for:

☑ Any Property type that has a set of different possible values (like enumeration, boolean, color, etc).

The examples given for customizing the possible values of checkbox and listbox feels also apply to the RadioButton feel.

6.3.13 FeelTrackbar and FeelTrackbarEdit

- Creates a trackbar to the right of an optional textbox.

Configuring a Property

At design-time

```
[PropertyFeel(PropertyGrid.FeelTrackbar)]
public int MyProperty {
    ...
```

At runtime

```
propEnum = AppendProperty(parentEnum, id, "Label", this, "MyProperty", "");
propEnum.Property.Feel = GetRegisteredFeel(PropertyGrid.FeelTrackbar);
```

Property types

The inplace control uses the .Net TrackBar control and therefore accepts only integers.

Setup

At both design-time and runtime, you can extend the trackbar's behaviour by configuring it the moment it appears. This is done when handling the *InPlaceCtrlVisibleEvent* event, by casting the inplace control to the correct type and calling some properties:

```
protected override void OnInPlaceCtrlVisible(InPlaceCtrlVisibleEventArgs e)
{
    if (e.PropertyEnum.Property.Id == myId)
        ((PropInPlaceTrackbar)e.InPlaceCtrl).SmallChange = 2;

    base.OnInPlaceCtrlVisible(e);
}
```

Some accessible properties

SmallChange	Accesses the SmallChange property of the .Net trackbar.

LargeChange	Accesses the LargeChange property of the .Net trackbar.
RealtimeChange	If true, modifying the trackbar immediately changes the value of the Property. If false, the value is validated only on leaving the control.
TrackBar	Returns a reference to the actual .Net trackbar control used internally.

Using a validator class to set limits

Validator classes can set limits on the trackbar (see "Validating the values" on page 53). SPG comes with a handy min/max validator for just this purpose. Assign it to the property and the feel will detect it:

```
[PropertyValidator(typeof(PropertyValidatorMinMax), 0, 200)]
public int MyProperty {
...
```

6.4 Handling the PropertyValue's value

6.4.1 Setting the value programmatically

Typically, the value of a property is edited by the end-user. However, you may want to do this programmatically, for example when dealing with managed Properties.

To set a value programmatically

```
propEnum.Property.Value.SetValue(myValue)
```

- *myValue* is a value of the property's type or whose type can be converted to the property's type.

To return the property's value

1) Use the following:

```
object value = propEnum.Property.Value.GetValue();
```

2) Cast it to the expected type.

6.4.2 Multiple value updates

Sometimes you need to update many values at the same time. This can cause the PropertyGrid to flicker because the control is refreshed by each update. You can prevent this by encapsulating all your updates in the following block:

```
myGrid.BeginUpdate();
...
myGrid.EndUpdate();
```

6.4.3 Testing for multiple values

When you use multiple target instances on a Property (by using *SelectedObjects* or by calling *PropertyGrid.AddTargetInstance*) non-unique values may result in the Property displaying a blank string.

To test for multiple values

```
bool valueUnicity = propEnum.Property.Value.HasMultipleValues;
```

6.4.4 Helper properties

There are several helper properties in the *PropertyValue* class:

Helper property	Returns
TargetInstance	Target instance attached to a Property (the first, if there are several).
PropertyDescriptor	PropertyDescriptor attached to a Property.
TypeConverter	Registered TypeConverter for the type of the Property’s value.

7 Validating the values

Microsoft PropertyGrid uses a *TypeConverter* to transform user-entered strings into values. If an exception is raised, it displays a generic popup form. The SPG framework, however, is able to catch three types of errors, enabling you to handle them as required:

- Exceptions raised by a *TypeConverter*
- Exceptions generated by the set accessor of the client application's property
- Failed validations resulting from a special Validator class

Each has an error code, which is transmitted to the client application when the *ValueValidation* event is raised (see *ValueValidation* on page 24).

7.1 Error codes

These are the error codes for *PropertyValue.ValueValidationResult*:

TypeConverterFailed	The failed validation originates from a *TypeConverter* that raised an exception.
ValidatorFailed	The *Check* method of the validator attached to the Property returned "False".
ExceptionByClient	An exception occurred in the client application code (inside a set accessor method).

The following are part of the same enumeration, and are used when the second, and subsequent, *ValueValidation* event is sent to the client application:

Validated	A subsequently entered value has been successfully validated.
PreviousValueRestored	A previously valid value has been restored.

7.2 Event handler

The event handler receives a ValueValidationEventArgs parameter. The fields enable the client application to determine what happened:

PropertyEnum	An enumerator to the property edited by the user.
InvalidPropEnum	An enumerator to the invalid property. This usually equals *RightBound*, indicating that the property referenced by *PropertyEnum* is the invalid one. However, it points to a child if its value has become invalid due to changes to the parent's value.
ValueValidationResult	One of the error codes explained in Error codes, above

ValueToValidate	The invalid value.
Exception	References the exception generated, if any.
Message	A string message set by the validator class, or equal to the message given by the exception.

7.3 Validation modes

The *PropertyGrid.ValueNotValidBehaviorMode* property enables the client application to choose between two modes from the *PropertyGrid.ValueNotValidBehaviorModes* enumeration:

IgnoreAndRestore	Any invalid value is instantly replaced by the previous valid value. The client application receives two notifications: the invalid code, and the *PreviousValueRestored* code.
KeepFocus	When the user tries to leave the inplace control containing an invalid value, the focus is forced on the control, and the proper notification is sent to the client application. In this mode, when an error code is sent, the client application can choose how to react, for example display a smart dialog, a simple message box, or play a beep. It can also show handy inplace tooltips (see “Skybound VisualTips” on page 55).

7.4 Validator classes

As discussed above, an error can originate from the check performed by a validator class derived from *PropertyValidatorBase*, for example the class *PropertyValidatorMinMax* used in the sample code.

The *PropertyValidatorAttribute* attribute can set a validator for a Property discovered by reflection. You can also set a validator at runtime.

To set a validator at runtime

*) Use this code:

```
propEnum.Property.Value.Validator = new PropertyValidatorMinMax(0, 300);
```

When writing your own validator

Override the virtual method contained by the *PropertyValidatorBase* class:

```
public virtual bool Check(object value, bool modify);
```

value	The value to be checked.
modify	If true, your code should call *PropertyValue.SetValue* with an appropriate value. This avoids initializing a Property with an invalid value.
Returns	Returns true if the value is valid, false otherwise.

7.5 Skybound VisualTips

SPG is compatible with VisualTips, free 3rd-party library from Skybound (www.skybound.ca). This supports the display of tips during value validation, and also "show help" or hovering tips. For further information, please see the implementation in the sample code (look into the OnValueValidation method), and also consult the latest VisualTips documentation.

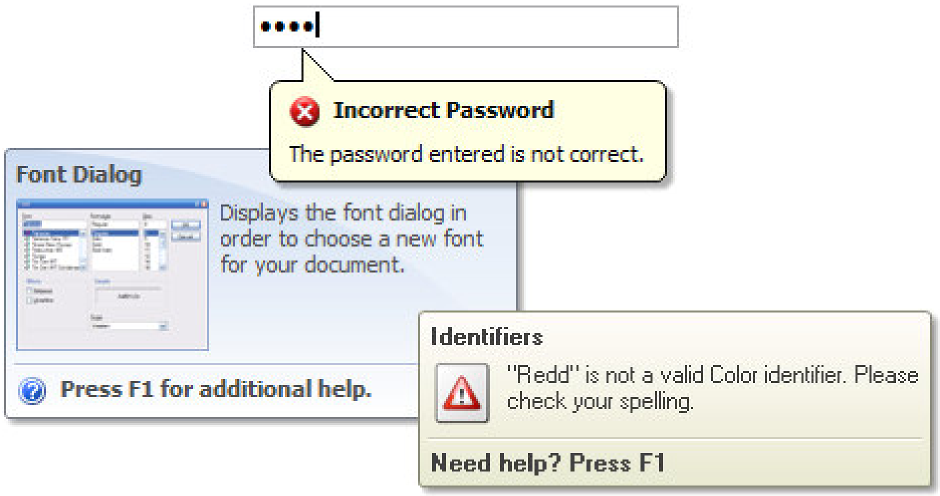

••••
Incorrect Password
The password entered is not correct.
Font Dialog
Displays the font dialog in order to choose a new font for your document.
Press F1 for additional help.
Identifiers
"Redd" is not a valid Color identifier. Please check your spelling.
Need help? Press F1

Figure 3 – Different VisualTips looks

8 Dynamic Properties

8.1 Enabled state

Any category and Property can be enabled and disabled at will. When a Property is disabled, no inplace control is shown on selection. Changing the Enabled state of a Property also changes the state of its descendents.

To return the Enabled state of a Property

*) Use the code:

```
bool enabled = propEnum.Property.Enabled;
```

To change the Enabled state

*) Use the code:

```
myGrid.EnableProperty(propEnum, true/false);
```

8.1.1 Displaying an enabled/disabled checkbox

It is possible to display a checkbox to the left of the Property label, allowing the end-user to enable and disable it and all its descendents.

The ancestor's enabled state always overrides the checkbox. For example a Property may have a checkbox set to true, but be disabled by a disabled parent.

To display an enabled/disabled checkbox

*) Use the code:

```
myGrid.SetManuallyDisabled(propEnum, true/false);
```

To get the boolean value of the enabled/disabled checkbox

*) Use the code:

```
bool checkboxValue = propEnum.Property.GetManuallyDisabledVariable();
```

To confirm that a Property has an enabled/disabled checkbox

*) Use the code:

```
bool check = propEnum.Property.CanBeDisabledManually;
```

If the checkbox is not present, the Enabled/Disabled state of the Property is returned.

8.2 Visible state

Any category and Property can be hidden or shown at any time. A Property with a collapsed ancestor, or positioned outside the current view of the grid will not be shown, regardless of whether Visible is true.

To return the visibility state of the Property

*) Use the following code

```
bool visible = propEnum.Property.Visible;
```

To show or hide a category or Property

*) Use the following call:

```
myGrid.ShowProperty(propEnum, true/false);
```

To force a Property to appear inside the current view of the grid

*) Use the following call:

```
myGrid.EnsureVisible(propEnum);
```

8.3 Expanded state

A category or a Property with children can be expanded and collapsed at will.

To return the expanded state of a Property

*) Use the code:

```
bool expanded = propEnum.Property.expanded;
```

To expand or collapse a category or Property

*) Use the call:

```
myGrid.ExpandProperty(propEnum, true/false);
```

To expand or collapse all of the grid's categories and Properties

*) Use the utility method:

```
myGrid.ExpandAllProperties(true/false);
```

8.4 Read-only state

A property can be switched to read-only at runtime. On selection, it displays either a disabled or read only text box depending on the value of *PropertyGrid.ReadOnlyVisual*.

To change the read only state of a Property

*) Use the following code:

```
propEnum.Property.Value.ReadOnly = true/false;
```

8.5 Selected state

Selecting a Property displays its inplace control, if it has one.

To return the selected state of a Property

*) Use the code

```
bool selected = propEnum.Property.Selected;
```

To select a category or Property

*) Use this call:

```
myGrid.SelectProperty(propEnum);
```

*) If you need to force the focus on the inplace control of a Property, use this call (the second parameter, if true, selects all the text):

```
myGrid.SelectAndFocusProperty(propEnum, true);
```

To ensure that no Property is selected

*) Use this call:

```
myGrid.SelectProperty(myGrid.RightBound);
```

8.6 Deleting a Property

Deleting a Property permanently deletes it and all its descendents.

To delete a Property

*) Use the call:

```
myGrid.DeleteProperty(propEnum);
```

9 Customizing the appearance

The appearance of the PropertyGrid can be customized in many ways, either globally to match your user interface, or on a Property-by-Property basis.

9.1 Global appearance

9.1.1 Drawing manager

The drawing manager draws most of the PropertyGrid. This class, derived from *CustomDrawManager*, applies a unified look with a certain style. Typically, a predefined drawing manager is used. However, you can create and assign your own instance.

To assign a predefined drawing manager

*) Use the code:

```
DrawingManager = DrawManagers.DotnetDrawManager;
```

To assign a custom instance of the drawing manager

*) Use the code:

```
DrawManager = new MyCustomDrawManager();
```

9.1.2 Layout and appearance

Several properties enable you to modify the global layout of the PropertyGrid plus appearance options not handled by the drawing manager.

Property	Sets or returns
CommentsBackColor	Background color of Comments area.
CommentsForeColor	Text color of Comments area
DisabledForeColor	Text color of disabled Properties. (Global.) Each Property can still be individually customized.
GridBackColor	Background color of grid (the area behind the Properties). Each Property can still be individually customized.
GridForeColor	Text color of every string displayed in the grid (labels and values). Each Property can still be individually customized.
GridColor	Color of the vertical and horizontal lines making up the grid.

Property	Sets or returns
HighlightedTextColor	Text color for the currently selected Property.
ReadonlyForeColor	Text color of any read-only property (label and value). Each Property can still be individually customized.
SelectedNotFocusedBkgColor	Background color of the selected Property when the PropertyGrid does not have the focus.
CommentsHeight	Height, in pixels, of the comments area.
CommentsVisibility	Visibility status of the comments area (boolean).
ToolbarVisibility	Visibility status of the toolbar (boolean).
Toolbar	A reference on the toolbar for customization of its content.
ToolTipMode	Options for the display of tooltips: ▪ *None* – none displayed. ▪ *ToolTipsOnLabels* - on labels only. ▪ *ToolTipsOnValues* - on values only. ▪ *ToolTipsOnValuesAndLabels* - on both labels and values.
EllipsisMode	Options for the display of the ellipsis (…) indicating a truncated label or value string: ▪ *None* - one displayed. ▪ *EllipsisOnLabels* - on labels only. ▪ *EllipsisOnValues* - on values only. ▪ *EllipsisOnValuesAndLabels* - on both labels and values.
Font	Font used to display every string in the grid. Each Property can still be customized individually.
GlobalTextMargin	Margin in pixels, calculated based on the active global font. This harmonizes the spacing between lines, icons and text. You can use this value when developing custom drawing managers or inplace controls.
LabelColumnWidth	Returns the width, in pixels, of the column containing the labels.
LabelColumnWidthRatio	The ratio between the label and value columns (0.0 to 1.0).
LeftColumnWidth	Width, in pixels, of the first column that hosts the +/- glyphs of the root categories.
PropertyVerticalMargin	Width, in pixels, of the margin applied above and below Property label and value texts.

Property	Sets or returns
ReadOnlyVisual	Options for the behavior of read-only Properties: ▪ *ReadOnlyFeel*: A textbox is displayed when the Property is active. Clipboard operation is enabled. ▪ *Disabled*: Like disabled Properties. No inplace control is shown.

9.1.3 Disabled state

SPG - unlike MSPG - has many options controlling the way the grid is disabled.

To disable the grid

*) Use the following code

```
Enabled = false;
myGrid.DisableMode = PropertyGrid.DisableModes. ...;
```

DisableModes are listed below:

Disable Modes

All of the following are exclusive:

None	Temporarily ignores having Enabled set to False.
Simple	Default mode. The whole grid is manageable, but all the textboxes are read-only. Clipboard operation is enabled.
NoValueSelection	Like *Simple*, except inplace controls are never shown and Clipboard operation is disabled.
NoPropertySelection	Like the *NoValueSelection* except that no Property can be selected. Nodes can be expanded and collapsed, and the grid scrolled.
Full	Nothing is selectable, and nothing operates. Like a disabled MSPG.

Additionally the *DisableModeGrayedOut* property, when set to true, ensures that all strings are grayed out. It can be paired with any of the above modes.

9.2 Property appearance

9.2.1 Property Class properties

Several properties of the Property class enable you to modify the appearance of individual Properties.

Property	**Sets or returns**
BackColor	Background color of the whole Property, label and value. (You can override this for the value by using *PropertyValue.BackColor* property.)
ForeColor	Text color of the whole Property, label and value. (You can override this for the value by using the *PropertyValue.ForeColor* property.)
DisabledForeColor	Text color of the Property when disabled, label and value. (You can override this for the value by using the *PropertyValue.DisabledForeColor* property.)
ReadOnlyForeColor	Text color of a readonly Property, label and value. (You can override this for the value by using the *PropertyValue.ReadOnlyForeColor* property.)
Font	Font for the Property's strings. (You can override this for the value by using the *PropertyValue.Font* property.)
DisplayName	Label for the Property.
Comment	Comment for the Property. The comment is displayed in the comment area.
ImageIndex	The index for an image from the *ImageList*, to be displayed to the left of a category, subcategory or property label. (The *ImageList* is set at the PropertyGrid level, so that it can be used by any property.)

9.2.2 PropertyValue class properties

The *PropertyValue* class contains additional properties for customizing its display:

Property	**Sets or returns**
ImageList	Place an icon in front of each entry. Used mainly by listboxes, but useful for displaying an image in front of a Property value.
ImageIndex	The index of an image in the *ImageList,* above. The image will be displayed on the left of the value string (and at the left of entries in the listboxes).
ImageSize	Image size in pixels. Normally, the image is drawn to fit the height of a Property. This option prevents the image from being stretched oddly if the Property height has been increased by the Font size, or by *PropertyVerticalMargin*.

10 Sorting the Properties

There are several ways to specify the sort order of the PropertyGrid. This may be necessary after using *SelectedObject(s)*, which results in categories and Properties being ordered according to the *TypeConverter* or *TypeDescriptor* classes.

10.1 Ordering on attributes

Applying the *SortedCategory* attribute or *the SortedProperty* attribute enables you to specify the sort order.

10.1.1 SortedCategory attribute

The PropertyGrid sorts parent categories by ascending order of the index placed in *SortedCategory* attribute (if applied instead of the usual *Category* attribute). Those without *SortedCategory* set appear below the rest, in order of discovery by reflection.

Example

This places the category “Appearance” in the 4th position amongst the root categories:

```
[SortedCategory("Appearance", 4)]
public int MyProperty1 {
    ...

[SortedCategory("Appearance", 4)]
public int MyProperty2 {
    ...
```

10.1.2 SortedProperty attribute

The *SortedProperty* attribute acts like *SortedCategory* (above), but for child Properties inside a parent category.

Example

This sorts three properties at the same level in the order *MyProperty3*, *MyProperty1* and *MyProperty2*:

```
[SortedProperty(2)]
public int MyProperty1 {
    ...

public int MyProperty2 {
    ...

[SortedProperty(1)]
public int MyProperty3 {
    ...
```

10.2 Using a PropertyComparer

Sometimes a complex sorting mechanism is required. Achieve this by assigning a custom *PropertyComparer*, derived from the *IComparer* class.

PropertyComparerDefaultSort class is the default, which sorts alphabetically.

The selected *PropertyComparer* compares *PropertyDescriptor* instances, so the code of your comparer class should look like this:

```
public class MyPropertyComparer : IComparer
{
    public int Compare(object x, object y)
    {
        PropertyDescriptor p1 = x as PropertyDescriptor;
        PropertyDescriptor p2 = y as PropertyDescriptor;
...
```

Assign it to the grid before calling *SelectedObject(s)*:

```
myGrid.PropertyComparer = new MyPropertyComparer();
myGrid.SelectedObject = myTargetInstance;
```

11 Keyboard management

11.1 Navigation

11.1.1 Default navigation

By default, the end-user edits and navigates using particular keys:

End-user edits and navigates like this:
1) Selects a Property
2) Presses ENTER to focus the inplace control.
3) Types a value.
4) Presses ENTER or ESC to terminate the editing.
5) Presses an arrow key (or HOME, PG DOWN, …) to go to the next property.

11.1.2 TAB key Navigation

Unlike MSPG, SPG supports navigation using the TAB key.

To enable TAB key navigation
*) Use the code:

```
NavigationKeyMode = NavigationKeyModes.TabKey;
```

Additional submodes (available with the TabKey*NavigationModes* enumeration) enable various options when *TabKey* is used:

TabKeyWithAutoFocus	Tabbing into a property gives the focus to its inplace control and selects any text in the Textbox.
TabKeyLeavesAtExtremes	Enable Shift+TAB and TAB out of the top and bottom of the PropertyGrid, transferring the focus to a next control in the parent container. Without this mode, TAB cycles indefinitely through Properties inside the grid.
TabKeyInSubControls	TAB through child controls inside the inplace control, before moving onto the next Property. (The default is to TAB from one Property to the next.)
TabKeyInChildProperties	Expand collapsed hierarchies as required to TAB through hidden child Properties.

TabKeySkipReadonlyProperties	Skip read-only Properties. Otherwise, read-only property can take the focus, enabling Clipboard Copy.
NoSubMode (default)	None of the above.

You can set this submode with the *PropertyGrid.TabKeyNavigationMode* property.

To restore default arrow key navigation

*) Use the code:

```
NavigationKeyMode = NavigationKeyModes.ArrowKeys;
```

11.2 Overriding keys

It is possible to override a key's default behavior by capturing the key before processing in *ProcessKeyPreview*.

Example

This intercepts the Delete key. Instead of giving the focus to a textbox and deleting its content, it deletes the whole Property:

```
protected override bool ProcessKeyPreview(ref Message m)
{
    if (m.Msg == 0x0100)
    {
        if ((Keys)m.WParam == Keys.Delete)
        {
            DeleteProperty(SelectedPropertyEnumerator);
            return true;
        }
    }

    return base.ProcessKeyPreview(ref m);
}
```