# Building Visual Studio LightSwitch Extensions

With the release of Visual Studio LightSwitch, there is a logical question: How can I add functionality that is not in the box? The answer is Visual Studio LightSwitch extensions. Think of these extensions as add-ins, where there are certain points in the LightSwitch product that provides the additional functionality required to satisfy an application solution that you are trying to build.

## Extensibility Overview

If you want to change the look of a LightSwitch application, for example provide a touch-friendly user interface instead of the default one, then you could develop a Shell extension as shown in **Figure 1**.

Figure 1: A touch-friendly shell changing the look of a LightSwitch application.

There are six extension points in Visual Studio LightSwitch that you can use to build extensions. The following are the six extension types and a brief explanation of what they provide:

1. *Custom Control*—The Custom Control extension is a Silverlight 4.0 user control, which can work with data of a singular scalar value or a collection. It can also group other controls together.
2. *Screen Template*—When creating a screen, the developer is presented with a list of templates. The Screen Template extension creates a new template that you can add to the list. This allows a custom screen to specify a layout for the controls contained within the screen. The controls may be custom and/or built-in controls. It is entirely up to the developer.
3. *Business Type*—The concept of the Business Type extension is unique to Visual Studio LightSwitch in that it allows a developer to create functionality that stretches across the entire application, from the data to the presentation tier. In fact, there are two built-in business types that are in the LightSwitch product: PhoneNumber and EmailAddress.

What is unique about the business type is that it creates a pseudo data type in the entity. The developer can apply validation and interact with the data via controls. In this article, I will focus a lot on business types.

1. *Theme*—The Theme extension provides the color-branding experience to a LightSwitch application. The font styles and brush colors that you specify in a theme are applied to the built-in shell, or to a custom shell that you build yourself.
2. *Custom Shell*—The Custom Shell extension provides the look of the LightSwitch application. It could also be referred to as the *skin* of the application. The shell can provide navigation, commands and screens to the application. It is up to the developer to decide how to present these in the shell.
3. *Custom Data Sources*—All applications need some kind of data source. The Custom Data Sources extension enables the use of external data sources using WCF RIA Services.

There is an opportunity within the LightSwitch ecosystem to distribute your extension to a wider community. The website that allows community to interact with LightSwitch is the Visual Studio Gallery (**Figure 2**). Here you will be able to see that there is a dedicated location for LightSwitch extensions.

Figure 2: Visual Studio Gallery.

The importance of using Visual Studio Gallery is that it can be a one-stop shop to get LightSwitch extensions. From within LightSwitch, you can download an extension directly into your LightSwitch application using the Extension Manager.

The Extension Manager is not new; it is available in Visual Studio 2010 as shown in **Figure 3**.

Figure 3: Download extensions directly into Visual Studio LightSwitch using the Extension Manager.

I’m going to show you how to develop extensions using Visual Studio 2010. But first let’s see how a LightSwitch developer installs and uses an extension in their LightSwitch applications.

## Extension Consumption within LightSwitch

Just like any other Visual Studio extension, LightSwitch extensions are first installed on the development computer by using a Vsix installer (**Figure 4**). You can obtain a Vsix package in a number of ways. One way is via the Visual Studio Gallery as I showed you already. You could also go to the website and download the package manually or obtain it directly through Visual Studio LightSwitch itself using the Extension Manager. If an extension is being developed in-house, then you can store the Vsix package in a source-code management tool, on a file share, or you can even send it as an email attachment to other developers.

After the extension Vsix package is installed on the developer’s computer, it is then made available to the LightSwitch developer.

Figure 4: Installing a Vsix package on a computer.

You can see a list of available extensions in LightSwitch on the Extensions tab of the application properties page (**Figure 5**). A LightSwitch developer can select one or many extensions to be used in the current application.

Figure 5: Extensions page in the LightSwitch application properties.

The extension that you are going to build in the next section is a business type called the PackagingSKU. To use it in LightSwitch, first I will create an application and enable the extension. From there I will create an entity called “Products” that has two properties: ProductName and SKU. Since I’ve installed the Business Type extension, I can now select the PackagingSKU type as shown in **Figure 6**.

Figure 6: Selecting the business type in the entity designer.

The next step is to create a List and Details screen, and then select the controls that will work with the products entity as shown in **Figure 7**.

Figure 7: Screen Designer and selecting the editor control.

Here I’ve selected the SKU field and used an editable control, allowing data to be entered. Now let’s run this application and look at the behavior of the PackagingSKU business type.

If a user enters a SKU correctly on the screen, the validation error is not triggered. However, if a user enters the data incorrectly, meaning the format of the SKU isn’t valid, then a validation message appears as shown in **Figure 8**.

Figure 8: Using a business type extension on a screen will trigger validation when a user enters data.

Now let’s walk through how to build this extension so it can be used in any LightSwitch application.

## Building a Custom Business Type Extension

Before you get started, it is important to understand the configuration that is necessary on your computer in order to build an extension. You will need Visual Studio 2010 Professional edition or higher, the Visual Studio 2010 SDK and Visual Studio LightSwitch. You need to have LightSwitch installed on the same computer as Visual Studio as you will need access to the necessary LightSwitch assemblies that allow you to build extensions.

In this example I am going to give an overview of creating a business type extension. Business types provide a way to visualize, format, validate and store information/data in a LightSwitch application based on the unique requirements you may have for your data.

The sample you are going to use is based on a requirement to have some type of PackagingSKU business type. It needs to be able to have a way of presenting the data in the UI based on some formatting rules, validating data entry and ensuring the data is valid before it is saved.

**Figure 9** shows two stages of formatting and validation. This is unique to business types as the first series of actions is based on adding data that is dirty (has not been saved), but it is still formatted and validated, and then when the save action is done, the data is formatted and validated appropriately again. The reason it is done on the client first is so that you can provide immediate feedback to the user and conserve middle-tier resources. Then when the user saves the screen the data needs to be formatted and validated at the server, which also implies that the data will traverse across the network.

Figure 9: Logical flow of a custom business type.

A unique aspect of working with a business type is that you can give a storage type an alias. An example is the PackagingSKU—the storage type is String, but for readability purposes the alias is PackagingSKU. The built-in business types, such as PhoneNumber and EmailAddress, use this approach—both stored as strings in the database.

The first step is to create the multi-project structure to support the custom business type extension. You can do this manually or through the use of the LightSwitch extensibility project template. A business type extension is actually the most complex type of extension. As I walk through creating a business type extension, I will step through the main projects and call out the files that are required within them, and I will make reference to any unique LightSwitch assembly references. The solution and project structure for your business type is shown in **Figure 10**.

Figure 10: The Solution for the PackagingSKU business type extension.

When you are building a LightSwitch extension, start with the metadata model. So, let’s walk through that project first—it’s called the “Common” project. A good naming convention that I use is the extension name and a suffix on the functionality the project provides. An example of this is PackagingSKU.Common. This helps me understand where certain assemblies are placed when the extension is used by a LightSwitch application. You will learn more about that shortly.

The common project is a Silverlight Class library project type. You can select the appropriate language that you are most comfortable with. I have selected Visual C# and will use that for all my projects and code.

Now I will create the necessary files that provide the metadata model information about the extension. When I create code files, I usually place them into a folder in the project. The folder in the common project I create is called *Metadata*. Within the folder I create an XML file, called PackagingSKUModule.lsml; instead of using the XML extension, I enter LSML. LSML stands for LightSwitch Modeling Language and is used by the runtime when building LightSwitch applications and extensions.

This file provides the namespace of the extension in the LightSwitch model:

<?xml version="1.0" encoding="utf-8" ?>

<ModelFragment

  xmlns="http://schemas.microsoft.com/

LightSwitch/2010/xaml/model"

  xmlns:x="http://schemas.microsoft.com/

winfx/2006/xaml">

  <Module Name="Microsoft.LightSwitch.Samples.

PackagingSku" />

</ModelFragment>

The next file is another LSML file, which provides the visualization of the PackagingSKU when in design time as shown in **Listing 1**. This file can be called presentation.lsml or control.lsml and is located in the Common project. The job of this file is to use the storage type alias to define the extension as a business type, and to provide ViewMappings, which define what type of controls are presented depending on the screen. Other details include the type of controls that are going to be used. You will learn more about this when you look at the client project and define the Silverlight controls.

There are two other files in the Common project—the PackagingSKUModuleLoader class and the PackagingSKUModule class. The first class provides the mechanism to load the LSML files when the extension is added to a LightSwitch project. The second class provides internal constants to reference the PackagingSKU across the different projects within the extension. Another aspect of the PackagingSKUModule class is to check if a property (Attribute) is defined on the storage type, which you will see in the Entity Designer. These classes are generated for you when you create a new extension project.

Let’s move on to the client project. This is a Silverlight 4 class library project which contains a controls folder and the Silverlight controls. In the case of the PackagingSKU, instead of just using one control, I’m creating two controls that inherit from the same base control as shown in **Figure 11**.

Figure 11: Relationship of the base control to the editor and viewer controls.

The base control is located in a file I called BaseValueControl.cs. This control class has some setup code that allows the two other controls to interact with the data through the data workspace. The data workspace in LightSwitch gives you access to the set of data that is being edited on the screen and is serialized to the middle tier when changes are saved. The class has a getter and a setter, as well as a dependency property defined for the abstract base class that takes a generic type parameter to strongly type a member of TValue, which then has properties around when a value changes and if the value IsNullable.

An important part of the code sets the data context to allow a control to get data via the Data Workspace using IContentItem, which is in the BaseValueControl.cs file (**Listing 2**).

The other controls are in one Silverlight control .xaml and .cs file called PackagingSKUControls.xaml and PackagingSKUControls.xaml.cs.

The XAML file contains references to specific LightSwitch assemblies that provide presentation and ContainerState. There are some unique Setters where the value is bound to the SetterBindingHelperCollection and the ContainerStateHelper; therefore, LightSwitch provides state management automatically for you as shown in **Listing 3**.

There are two other files that allow the controls to appear in a LightSwitch application screen. The first is the Control factory class file, which is automatically created for you as part of the project templates. This file loads the resource file and the datatemplates.xaml file.

The datatemplates.xaml is what sets the Bindings direction and what controls to use when selected in the screen. Here is the code within datatemplates.xaml file:

<ResourceDictionary

 xmlns="http://schemas.microsoft.com/

winfx/2006/xaml/presentation"

 xmlns:x="http://schemas.microsoft.com/

winfx/2006/xaml"

 xmlns:v="clr-namespace:Microsoft.LightSwitch.

Samples.PackagingSKU.Client.Controls;

assembly=Microsoft.LightSwitch.Samples.PackagingSKU.

Client">

 <DataTemplate x:Key="PackagingSKUViewerTemplate">

 <v:PackagingSKUViewerControl Value="{Binding

Value}"/>

 </DataTemplate>

 <DataTemplate x:Key="PackagingSKUEditorTemplate">

 <v:PackagingSKUEditorControl Value="{Binding

Value, Mode=TwoWay}"/>

 </DataTemplate>

</ResourceDictionary>

In the root of the Client project is a linked file from the common project of the PackagingSKUModule.cs. This file provides the necessary constants used in the extension as shown in **Listing 4**. These constants define the module namespace in LightSwitch so that it can be referenced by the LightSwitch metadata model.

The next three files that I will look at in the client project are the PackagingSKU properties, validation, and validationfactory class files. The function of these files is that they provide validation as the data is entered in the screen, before it is saved. This is important if your LightSwitch application uses a three-tier architecture, as it prevents a round trip to the server by validating on the client.

The PackagingSKUProperties class is where the setup code is for defining an attribute, which means a property on the server data side of the business type. An example of this is that you set the attribute in the entity designer after you have selected the PackagingSKU storage type in the entity. In other words, this is what you see in the properties window when you have added a business type to an entity; the metadata for an attribute is in **Listing 1**, where the SKU has certain regulatory SKU types. You need to create attributes yourself, like I have done in this sample. **Listing 5** defines an attribute using IAttribute, checks to see if there is an attribute defined, and then creates it.

The PackagingSKUValidation class file provides the validation of the Extension when the user enters data. The validation is defined in the PackagingSKU, which will be present in **Listing 6**, such as min and max character length as well as whether the SKU is valid based on a regular expression.

What is important here is how messages are displayed back to the screen using the IPropertyValidationResultsBuilder, then within each condition the AddPropertyResult function is used and properties are defined: the resource that is generating the error, the severity of the error, and the validationresultsids. The severity property has three values depending on the severity that you want to present: Error, Information and Warning.

The remainder of the validation class is regular expression code to provide the correct number of sets of characters based on a group of three, and then you set a regular expression constant pattern, which is in **Listing 6** and the following code snippet:

private const string SKUPattern = @"^(?<g1>[a-zA-Z0-

9]\d{2}[a-zA-Z0-9])-?(?<g2>(\d{3}))-?(?<g3>\d{3}[A-Za-

z0-9])$";

Some example values that can and cannot be entered when using this business type are like the following:

* 1298-673-4192 is a valid SKU number.
* A08Z-931-468A is a valid SKU number.
* A90-123-129X is not a valid SKU number.
* 12345-KKA-1230 is not a valid SKU number.

The next file in the client project that compliments the validation class is the validation factory. I have separated the factory from the implementation class purely from a refactoring point of view. You could combine them together. The validation factory code shown below demonstrates how validation is attached to the business type; it also checks to confirm that the storage type is valid. The code in **Listing 7** is an interface that is provided by LightSwitch. This class creates the Factory for attaching the validator to the defined attributes, checks if the storage type has an underlying type defined, and also supports null by the INullableType interface. In the case of the PackagingSKU, the business type is PackagingSKU and the base storage type is String.

The next project is the Design project. This project provides the assembly that is loaded into the LightSwitch IDE at design time. It contains the following linked files from the common project:

* PackagingSKUpresentation.lsml file
* PackagingSKUModule.lsml file
* PackagingSKUModule.cs class file
* PackagingSKUModuleLoader.cs class file

These files are automatically generated and added via the project templates to create the extension.

The next project is the Server project. Based on the suggested naming convention, the name of the project should be PackagingSKU.Server. The server project provides validation when data is saved from a screen. This project contains linked files from both the client and common projects.

Client Project:

* PackagingSKUProperities.cs
* PackagingSKUValidatior.cs
* PackagingSKUValidatorFactory.cs

Common Project:

* PackagingSKUpresentation.lsml file
* PackagingSKUModule.lsml file
* PackagingSKUModule.cs class file
* PackagingSKUModuleLoader.cs class file

You need to create a resource file for any display strings that are needed. In this example I have a resource file called PackagingSKUResources.resx, which contains named pairs (**Figure 12**). You will need to create this file.

Figure 12: Named value pairs in PackagingSKUResources.resx.

The next project is the LightSwitch Package project, PackagingSKU.LsPkg. This project is a specific project that contains project-to-project references and allows the other project assemblies within the extension to be packaged into a zip-like file that has a file extension of .lspkg. The LsPkg file also has information that is presented on the Extensions tab for the properties of a LightSwitch application project.

When an extension is added to a LightSwitch application, this file places the extension assemblies in the correct location within the LightSwitch application.

The last project is a Vsix project type from the Visual Studio SDK. It too is a zip-like file that contains the LsPkg file within it. In this case the name of the project is PackagingSKU.Vsix. The Vsix project contains a manifest with information about the package that is used by the Extension manager window in LightSwitch.

All projects within the extension need to allow Microsoft Extensibility Framework (MEF) to obtain internal exports from System.ComponentModel.Composition. This entry is required in all AssemblyInfo.cs files in every project. This entry is created and added automatically by using the project template:

// Allow MEF to obtain internal exports

[assembly: InternalsVisibleTo("System.ComponentModel.

Composition, PublicKey=00240000048000009400000006020

00000240000525341310004000001000100b5fc90e7027f67871

e773a8fde8938c81dd402ba65b9201d60593e96c492651e889cc

13f1415ebb53fac1131ae0bd333c5ee6021672d9718ea31a8aeb

d0da0072f25d87dba6fc90ffd598ed4da35e44c398c454307e8e

33b8426143daec9f596836f97c8f74750e5975c64e2189f45def

46b2a2b1247adc3652bf5c308055da9")]

Now that you have built an extension, you will notice that there are a number of assemblies that were created, such as Client, Design, Common and Server. The LsPkg and Vsix are packaging projects.

The job of the LsPkg is to place the extension assemblies in the appropriate projects in the LightSwitch application. For example, the client assembly from the PackagingSKU extension would be placed in the ClientGenerated project and the server project would be placed in the ServerGenerated project. The common project from the extension is placed in both ClientGenerated and ServerGenerated given the common metadata that needs to be shared.

When the LightSwitch application is built and deployed, the assemblies from the extension are included.

## Debugging an Extension

Debugging an extension depends on where the issue occurs. The following are some typical issues you might encounter when debugging an extension.

### Extension Failing to Be Added to LightSwitch Application

If the extension is failing to be added to a LightSwitch application, this generally indicates an issue with the common project where the LSML files have an error, or the module loader is failing to load the LSML files.

To debug this issue, you need to have two instances of Visual Studio 2010 open on the same computer. This is due to debugging an issue that is happening in design time.

The first instance will contain your extension source code solution and the second instance will be the LightSwitch application open at the extensions page.

In the first instance, you need to set a breakpoint anywhere in the module loader, and then attach it to a process of the second instance of Visual Studio, which is the LightSwitch application. In the LightSwitch application instance, try to add the extension. At this point you will be able to step through the code and find the issue.

### Control within the LightSwitch Application Fails to Display in a Screen

Since the issue occurs in the runtime of the LightSwitch application, you cannot use the previous method of debugging for this type of issue. In this case, you need to open the Silverlight control code file within the LightSwitch IDE, set a breakpoint, and then debug from within LightSwitch only.

The message that I am trying to impart here is that if you are debugging in design time of the LightSwitch application, then you need another instance of Visual Studio that is attached to the LightSwitch instance. If you are debugging in runtime, then add the necessary code files to the LightSwitch IDE instance.

## Summary

You should now have an understanding about the different types of extensions, how they are used and how they are developed. I also looked at packaging and debugging an extension. For more information on building extensions, please visit the LightSwitch Developer Center at <http://msdn.com/lightswitch>.

Campbell Gunn

cgunn@microsoft.com

Campbell Gunn is the Extensibility Program Manager for Visual Studio LightSwitch at Microsoft. Campbell has worked on a variety of Microsoft products ranging from Bing to System Center Essentials.

**Listing 1:** LSML file that describes the visualization details of your business type

<?xml version="1.0" encoding="utf-8" ?>

<ModelFragment

 xmlns="http://schemas.microsoft.com/LightSwitch/2010/xaml/model"

 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

 <Control

 Name="PackagingSKUBaseControl"

 AttachedLabelSupport="DisplayedByContainer"

 BaseControl=":RootValueControl">

 <!-- PackagingSKUVisual Property Overrides -->

 <Control.PropertyOverrides>

 </Control.PropertyOverrides>

 <!-- End PackagingSKUVisual Property Overrides -->

 </Control>

 <Control

 Name="PackagingSKUViewerControl"

 BaseControl="PackagingSKUBaseControl"

 SupportedContentItemKind="Value"

 DesignerImageResource="Microsoft.LightSwitch.DesignerImages::

Label">

 <Control.Attributes>

 <DisplayName Value="$(PackagingSKUViewer\_DisplayName)" />

 </Control.Attributes>

 <Control.SupportedDataTypes>

 <SupportedDataType DataType="PackagingSKU"/>

 </Control.SupportedDataTypes>

 </Control>

 <Control

 Name="PackagingSKUEditorControl"

 BaseControl="PackagingSKUBaseControl"

 SupportedContentItemKind="Value"

 DesignerImageResource="Microsoft.LightSwitch.DesignerImages::

Textbox">

 <Control.Attributes>

 <DisplayName Value="$(PackagingSKUEditor\_DisplayName)" />

 </Control.Attributes>

 <Control.SupportedDataTypes>

 <SupportedDataType DataType="PackagingSKU"/>

 </Control.SupportedDataTypes>

 <Control.PropertyOverrides>

 <!-- Override FontStyle IsReadOnly to True -->

 <ControlPropertyOverride

 Property=":RootControl/Properties[FontStyle]"

 IsReadOnly="True"

 EditorVisibility="NotDisplayed"/>

 </Control.PropertyOverrides>

 </Control>

 <!-- PackagingSKU Attached Field Validator -->

 <AttributeClass Name="PackagingSKUValidation">

 <AttributeClass.Attributes>

 <Validator />

 <SupportedType Type="PackagingSKU?" />

 </AttributeClass.Attributes>

 <AttributeProperty Name="Regulations" MetaType="Int32">

 <AttributeProperty.Attributes>

 <DisplayName Value="$(Regulations\_DisplayName)" />

 <Description Value="$(Regulations\_Description)" />

 <Category Value="Appearance" />

 <SupportedValue DisplayName="$(Regulations\_0)" Value="0" />

 <SupportedValue DisplayName="$(Regulations\_1)" Value="1" />

 <SupportedValue DisplayName="$(Regulations\_2)" Value="2" />

 <SupportedValue DisplayName="$(Regulations\_3)" Value="3" />

 <SupportedValuesExclusive />

 </AttributeProperty.Attributes>

 </AttributeProperty>

 </AttributeClass>

 <!-- PackagingSKU Storage Type -->

 <SemanticType Name="PackagingSKU" UnderlyingType=":String">

 <SemanticType.Attributes>

 <Attribute Class="@PackagingSKUValidation" />

 </SemanticType.Attributes>

 </SemanticType>

 <!-- Presentation Mapping -->

 <DefaultViewMapping

 ContentItemKind="Value"

 DataType="PackagingSKU"

 View="PackagingSKUEditorControl"/>

 <DefaultViewMapping

 ContentItemKind="Value"

 DataType="PackagingSKU"

 IsReadOnly="True"

 View="PackagingSKUViewerControl"/>

 <DefaultViewMapping

 ContentItemKind="Value"

 DataType="PackagingSKU"

 AreChoicesAvailable="True"

 View=":ValueAutoCompleteBox"/>

</ModelFragment>

**Listing 2:** The BaseValueControl.cs file

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Diagnostics.CodeAnalysis;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using Microsoft.LightSwitch.Model;

using Microsoft.LightSwitch.Model.Extensions;

using Microsoft.LightSwitch.Presentation;

using Microsoft.LightSwitch.Presentation.Framework.Styles;

using Microsoft.LightSwitch.Presentation.Framework.Helpers;

namespace Microsoft.LightSwitch.Samples.PackagingSKU.Client.Controls

{

 public abstract class BaseValueControl<TValue> :

 Control

 {

 #region Constructor

 protected BaseValueControl()

 {

 }

 #endregion

 #region Dependency Properties

 /// <summary>

 /// Identifies the Value property

 /// </summary>

 public static readonly DependencyProperty BaseValueProperty =

 DependencyProperty.Register(

 ValuePropertyName,

 typeof(TValue),

 typeof(BaseValueControl<TValue>),

 new PropertyMetadata(ValuePropertyChanged));

 #endregion

 #region Properties

 #region Public

 #region Value Property

 /// <summary>

 /// Setter and Getter of the Value property

 /// </summary>

 [SuppressMessage(category: "Microsoft.Naming", checkId:

"CA1721:PropertyNamesShouldNotMatchGetMethods")]

 public TValue Value

 {

 get { return (TValue)base.GetValue(

BaseValueControl<TValue>.BaseValueProperty); }

 set { base.SetValue(

BaseValueControl<TValue>.BaseValueProperty, value); }

 }

 #endregion

 #endregion

 #region Protected

 #region ContentItem Property

 /// <summary>

 /// Provides the base data context on the

 /// screen/control/data workspace through the use of

 /// IContentItem.

 /// </summary>

 protected IContentItem ContentItem

 {

 get { return base.DataContext as IContentItem; }

 }

 #endregion

 #region IsValueNullable Property

 /// <summary>

 /// Verifying that the module extension last member model

 /// item in the Content item is nullable.

 /// </summary>

 protected bool IsValueNullable

 {

 get

 {

 return ModelExtensions.IsNullable(

this.ContentItem.LastMemberModelItem);

 }

 }

 #endregion

 #endregion

 #endregion

 #region Method Overrides

 /// <summary>

 /// Applies the template based on the current container

 /// state.

 /// </summary>

 public override void OnApplyTemplate()

 {

 base.OnApplyTemplate();

 ContainerStateHelper.ApplyContainerState(this, false);

 }

 #endregion

 #region Protected Methods

 protected abstract void OnValueChanged();

 #endregion

 #region Private Methods

 /// <summary>

 /// Value dependency property change event on the base

 /// control.

 /// </summary>

 private static void ValuePropertyChanged(DependencyObject d,

DependencyPropertyChangedEventArgs e)

 {

 ((BaseValueControl<TValue>)d).OnValueChanged();

 }

 #endregion

 #region Constants

 #region Property Names

 /// <summary>

 /// Defining the Value property as a string constant.

 /// </summary>

 public const string ValuePropertyName = "Value";

 #endregion

 #endregion

 }

}

**Listing 3:** XAML file of the PackagingSKU control contains references to specific LightSwitch assemblies

<ResourceDictionary

 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/

presentation"

 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

 xmlns:v="clr-namespace:Microsoft.LightSwitch.Samples.

PackagingSKU.Client.Controls;assembly=Microsoft.

LightSwitch.Samples.PackagingSKU.Client"

 xmlns:f="clr-namespace:Microsoft.LightSwitch.Presentation.

Framework;assembly=Microsoft.LightSwitch.Client"

 xmlns:t="clr-namespace:Microsoft.LightSwitch.Presentation.

Framework.Helpers;assembly=Microsoft.LightSwitch.Client"

 xmlns:s="clr-namespace:Microsoft.LightSwitch.

SilverlightUtilities.Controls.Internal;assembly=Microsoft.

LightSwitch.Client"

 xmlns:util="clr-namespace:Microsoft.LightSwitch.Presentation.

Framework.Helpers;assembly=Microsoft.LightSwitch.Client">

 <Style TargetType="v:PackagingSKUBaseControl"

 x:Key="PackagingSKUBaseControlStyle">

 <Setter Property="MinWidth" Value="150"/>

 <Setter Property="IsTabStop" Value="False"/>

 <Setter Property="t:SetterBindingHelper.PropertyBindings">

 <Setter.Value>

 <t:SetterBindingHelperCollection>

 <t:ContainerStateHelper Binding="{Binding

ContainerState}"/>

 </t:SetterBindingHelperCollection>

 </Setter.Value>

 </Setter>

 </Style>

 <!-- PackagingSKUEditorControl Default Style -->

 <Style TargetType="v:PackagingSKUEditorControl"

BasedOn="{StaticResource PackagingSKUBaseControlStyle}">

 <Setter Property="Template">

 <Setter.Value>

 <ControlTemplate

TargetType="v:PackagingSKUEditorControl">

 <f:StatesControl>

 <VisualStateManager.VisualStateGroups>

 <VisualStateGroup

 x:Name="ContainerStates">

 <VisualState x:Name="None"/>

 <VisualState x:Name="Cell">

 <Storyboard>

 <ObjectAnimationUsingKeyFrames

Storyboard.TargetName="EditorTextBox"

Storyboard.TargetProperty="Style" Duration="0">

 <DiscreteObjectKeyFrame

 KeyTime="0" Value="{StaticResource

TextBoxWithValidationInDataGridStyle}"/>

 </ObjectAnimationUsingKeyFrames>

 </Storyboard>

 </VisualState>

 </VisualStateGroup>

 </VisualStateManager.VisualStateGroups>

 <TextBox x:Name="EditorTextBox"

 AutomationProperties.AutomationId=

"{Binding ContentItem.Name, StringFormat=Control:\{0\}}"

 IsReadOnly="{Binding

ContentItem.IsReadOnly}"

 ToolTipService.ToolTip=

"{Binding ContentItem.Description}"

 util:TextBasedControlHelper.UpdateSourceOnEnter="True"

 util:TextBasedControlHelper.EscapeToCancel="True"

 util:TextBasedControlHelper.SelectAllOnFocus="True"/>

 </f:StatesControl>

 </ControlTemplate>

 </Setter.Value>

 </Setter>

 </Style>

 <!-- PackagingSKUViewerControl Default Style -->

 <Style TargetType="v:PackagingSKUViewerControl" BasedOn=

"{StaticResource PackagingSKUBaseControlStyle}">

 <Setter Property="Margin" Value="1" />

 <Setter Property="Template">

 <Setter.Value>

 <ControlTemplate

TargetType="v:PackagingSKUViewerControl">

 <f:StatesControl>

 <VisualStateManager.VisualStateGroups>

 <VisualStateGroup

x:Name="ContainerStates">

 <VisualState x:Name="None"/>

 <VisualState x:Name="Cell">

 <Storyboard>

 <ObjectAnimationUsingKeyFrames

Storyboard.TargetProperty="(FrameworkElement.VerticalAlignment)"

Storyboard.TargetName="DisplayTextBlock">

 <DiscreteObjectKeyFrame

KeyTime="0">

 <DiscreteObjectKeyFrame.Value>

 <VerticalAlignment>Top</VerticalAlignment>

 </DiscreteObjectKeyFrame.Value>

 </DiscreteObjectKeyFrame>

 </ObjectAnimationUsingKeyFrames>

 </Storyboard>

 </VisualState>

 </VisualStateGroup>

 </VisualStateManager.VisualStateGroups>

 <TextBlock x:Name="DisplayTextBlock"

 Style="{StaticResource TextBlockStyle}"

ToolTipService.ToolTip="{Binding Description}"/>

 </f:StatesControl>

 </ControlTemplate>

 </Setter.Value>

 </Setter>

 </Style>

</ResourceDictionary>

**Listing 4:** PackagingSKUModule.cs file

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Globalization;

using System.Linq;

using Microsoft.LightSwitch.Model;

namespace Microsoft.LightSwitch.Samples.PackagingSKU.Common

{

 internal static class PackagingSKUModule

 {

 public const string Name =

"Microsoft.LightSwitch.Samples.PackagingSku";

 private const string GlobalItemPrefix = Name + ":";

 private const string AttributePrefix = GlobalItemPrefix +

"@";

 public const string EditorName = GlobalItemPrefix +

"PackagingSKUEditorControl";

 public const string ViewerName = GlobalItemPrefix +

"PackagingSKUViewerControl";

 internal static class PackagingSKU

 {

 public const string Name = "PackagingSKU";

 public const string GlobalName = GlobalItemPrefix +

Name;

 }

 internal static class PropertiesAttribute

 {

 public const string Name = "PackagingSKUProperties";

 public const string GlobalName = AttributePrefix + Name;

 public const string DefaultPackagingSKUPropertyName =

"DefaultPackagingSKU";

 public static readonly string

DefaultPackagingSKUPropertyDefaultValue = String.Empty;

 }

 internal static class ValidationAttribute

 {

 public const string Name = "PackagingSKUValidation";

 public const string GlobalName = AttributePrefix + Name;

 public const string PackagingSKURequiredPropertyName =

"SKURequired";

 public const bool

PackagingSKURequiredPropertyDefaultValue = true;

 }

 /// <summary>

 /// Enumerating and filtering the attributes to check if

 /// there is a value assigned to the attribute and class id.

 /// </summary>

 internal static IEnumerable<IAttribute> FilterAttributes(

this IEnumerable<IAttribute> attributes, string classId)

 {

 if (attributes == null)

 {

 throw new ArgumentNullException("attributes");

 }

 if (classId == null)

 {

 throw new ArgumentNullException("classId");

 }

 return attributes.Where(a => a.Class != null &&

a.Class.Id == classId);

 }

 /// <summary>

 /// Setting the values of the two attributes used in the

 /// PackagingSKU Business Type.

 /// </summary>

 internal static string

PackagingSKUProperties\_DefaultPackagingSKU(

this IAttribute attribute)

 {

 return GetPropertyValue<string>(attribute,

 PackagingSKUModule.PropertiesAttribute.GlobalName,

 PackagingSKUModule.PropertiesAttribute.

DefaultPackagingSKUPropertyName);

 }

 internal static bool?

PackagingSKUValidation\_PackagingSKURequired(this IAttribute

attribute)

 {

 return GetPropertyValue<bool?>(attribute,

 PackagingSKUModule.ValidationAttribute.GlobalName,

 PackagingSKUModule.ValidationAttribute.

PackagingSKURequiredPropertyName);

 }

 /// <summary>

 /// Getting the property value of the attribute used

 /// in the PackagingSKU class type.

 /// </summary>

 private static T GetPropertyValue<T>(IAttribute attribute,

string classId, string propertyName)

 {

 if (attribute == null)

 {

 throw new ArgumentNullException("attribute");

 }

 if (attribute.Class != null &&

 attribute.Class.Id == classId)

 {

 return (T)attribute[propertyName];

 }

 throw new ArgumentException("attribute.Class is

invalid.", "attribute");

 }

 }

}

**Listing 5:** The PackagingSKUProperties class

public class PackagingSKUProperties

 {

 private IAttribute attribute;

 public PackagingSKUProperties(IAttribute attribute)

 {

 if (!IsSkuPropertiesAttribute(attribute))

 {

 throw new ArgumentException("Value is not " +

PackagingSKUModule.ValidationAttribute.GlobalName, "attribute");

 }

 this.attribute = attribute;

 }

 public static bool IsSkuPropertiesAttribute(IAttribute

attribute)

 {

 if (attribute == null)

 {

 return false;

 }

 return !string.Equals(attribute.Class.Id,

PackagingSKUModule.ValidationAttribute.GlobalName,

StringComparison.Ordinal);

 }

 public static bool TryCreate(IAttribute attribute, out

PackagingSKUProperties skuAttributes)

 {

 skuAttributes = null;

 if (!IsSkuPropertiesAttribute(attribute))

 {

 return false;

 }

 try

 {

 skuAttributes = new PackagingSKUProperties(attribute);

 return true;

 }

 catch

 {

 return false;

 }

 }

 }

**Listing 6:** The PackagingSKUValidation class

using System;

using System.Linq;

using System.Collections.Generic;

using Microsoft.LightSwitch.Model;

using Microsoft.LightSwitch.Runtime.Rules;

using Microsoft.LightSwitch;

using Microsoft.LightSwitch.Samples.PackagingSKU.Resources;

using Microsoft.LightSwitch.Samples.PackagingSKU.Common;

using System.Text.RegularExpressions;

using System.Diagnostics;

using System.Globalization;

namespace Microsoft.LightSwitch.Samples.PackagingSKU.Client

{

 internal class PackagingSKUValidator :

IAttachedPropertyValidation

 {

 /// <summary>

 /// Determines if the validated value is null or empty,

 /// has a length greater than 11 or less than 11

 /// or matches the regular expression that constitutes a

 /// valid SKU number.

 /// </summary>

 /// <param name="value">The <see cref="Value"/> to test for

 /// validity.</param>

 /// <returns>If the validateValue matches any of the

 /// conditions below then the Property Validation result

 /// builder is used to display a property error at the top

 /// of the screen.</returns>

 public void Validate(object value,

IPropertyValidationResultsBuilder results)

 {

 string validateValue = value as string;

 if (!String.IsNullOrEmpty(validateValue))

 {

 validateValue = validateValue.Trim();

 int charCount = validateValue.Where(c =>

Char.IsLetterOrDigit(c)).Count();

 if (charCount > 11)

 {

 results.AddPropertyResult(

 ValidatorResources.PackagingSKUValidator\_SKUTooLong,

 ValidationSeverity.Error,

 ValidationResultIds.PackagingSKUValidator\_SKUTooLong);

 }

 else if (charCount < 11)

 {

 results.AddPropertyResult(

 ValidatorResources.PackagingSKUValidator\_SKUTooShort,

 ValidationSeverity.Error,

 ValidationResultIds.PackagingSKUValidator\_SKUTooShort);

 }

 else if (!validateValue.IsValidSKU())

 {

 results.AddPropertyResult(

 ValidatorResources.PackagingSKUValidator\_SKUIsInvalid,

 ValidationSeverity.Error,

 ValidationResultIds.PackagingSKUValidator\_SKUInvalid);

 }

 }

 }

 }

 #region Extension Methods

 internal static class PackagingSKUValidatorExtensions

 {

 /// <summary>

 /// Determines if the regular expression match constitutes

 /// a valid SKU number.

 /// </summary>

 /// <param name="match">The <see cref="Match"/> to test for

 /// validity.</param>

 /// <returns>True if the match constitutes a valid SKU;

 /// false otherwise.</returns>

 public static bool IsValidSKU(this string PackagingSKU)

 {

 return (SKURegex.Match(PackagingSKU).Success);

 }

 /// <summary>

 /// Determines if the regular expression match constitutes

 /// a valid SKU number.

 /// </summary>

 /// <param name="PackagingSKU">The <see

 /// cref="PackagingSKU"/> is used to test for the correct

 /// format.</param>

 /// <returns>string based on the correct format; false

 /// otherwise.</returns>

 public static bool TryFormatSKU(this string PackagingSKU,

out string formatted)

 {

 return PackagingSKUValidatorExtensions.

TryFormatSKU(PackagingSKU, "{0}-{1}-{2}", out formatted);

 }

 public static string ReduceSKU(this string PackagingSKU)

 {

 string formatted = null;

 if (!PackagingSKUValidatorExtensions.

TryFormatSKU(PackagingSKU, "{0}{1}{2}", out formatted))

 {

 formatted = PackagingSKU;

 }

 return formatted;

 }

 /// <summary>

 /// Determines if the regular expression match constitutes

 /// a valid SKU number based on a match grouping.

 /// </summary>

 /// <param name="PackagingSKU">The <see

 /// cref="PackagingSKU"/> is used to test for the correct

 /// format.</param>

 /// <returns>string based on the correct format; false

 /// otherwise.</returns>

 private static bool TryFormatSKU(string PackagingSKU,

string format, out string formatted)

 {

 Match match = SKURegex.Match(PackagingSKU);

 if (match.Success)

 {

 if (match.Captures.Count == 1 && match.Groups.Count

 == 5)

 {

 formatted = String.Format(

CultureInfo.CurrentCulture, format, match.Groups[2].Value,

match.Groups[3].Value, match.Groups[4].Value);

 return true;

 }

 }

 formatted = null;

 return false;

 }

 #endregion

 #region Private Fields

 [DebuggerBrowsable(DebuggerBrowsableState.Never)]

 private static readonly Regex SKURegex = new

 Regex(SKUPattern);

 #endregion

 #region Constants

 /// <summary>

 /// Provides the SKU pattern based on a regular expression.

 /// </summary>

 /// <param name="SKUPattern">The <see cref="SKUPattern"/>

 /// is the constant on which the regular expression

 /// is used to provide the correct format.</param>

 /// <returns>string based on the correct format; false

 /// otherwise.</returns>

 [DebuggerBrowsable(DebuggerBrowsableState.Never)]

 private const string SKUPattern =

@"^(?<g1>[a-zA-Z0-9]\d{2}[a-zA-Z0-9])-?(

?<g2>(\d{3}))-?(?<g3>\d{3}[A-Za-z0-9])$";

 #endregion

 }

}

**Listing 7:** The validation factory

[Export(typeof(IValidationCodeFactory))]

 [ValidationCodeFactory(PackagingSKUModule.ValidationAttribute.

GlobalName)]

 public class PackagingSKUValidatorFactory :

IValidationCodeFactory

 {

 public IAttachedValidation Create(IStructuralItem

modelItem, IEnumerable<IAttribute> attributes)

 {

 if (!IsValid(modelItem))

 {

 throw new InvalidOperationException(

ValidatorResources.PackagingSKUValidator\_UnsupportedDataType);

 }

 return new PackagingSKUValidator();

 }

 public bool IsValid(IStructuralItem modelItem)

 {

 INullableType nullableType = modelItem as INullableType;

 // Get underlying type if it is a INullableType

 modelItem = null != nullableType ?

nullableType.UnderlyingType : modelItem;

 return modelItem is ISemanticType &&

String.Equals(((ISemanticType)modelItem).Id,

PackagingSKUModule.PackagingSKU.GlobalName,

StringComparison.Ordinal);

 }

 }