A plugins that shows a custom dialog

Creating a plugin that displays a custom dialog

To run this example you must first create a C# project that refrecnes FTC.SDKInterface.dll, Feat.UIComposition.dll and Feat.Core.dll. The resulting dll from your project (along with any non .NET dependecies) must be placed in the Plugins directory in the FTC install location.

The Plugin class.

The first step is to create a class that implements the IPluginService interface. This is the type that will be instantiated by Scene Composer.

```
public class PluginWithMenuAndDialogPlugin : IPluginService
{

}
```

Implementing IPlugin

Next we must implement the IPluginService interface members.

```
public string Description
{
  get { return "A plugin with a menu item and dialog"; }
}

public string Name
{
  get { return this.GetType().Name; }
}

public string Version
{
  get { return "1.1.1"; }
}

public IEnumerable<int> FeatureIds
{
   get { return null; }
}
```

The initliaze method will register a menu command and a custom dialog Registering Commands: To register a command first an interface that derives from ICommandContainer must be created. The interface will contain properties for each of the commands that will be registered. The interface will be implemented internally by Scene Composer. The plugin can then register the commands using the UICommandManager service. This can be done either directly or using the helper class CommandDefinitionCollection that simplifies the registering of multiple commands.

The command can be invoked from code, (see other more complex examples for a demonstration), or can be displayed in the menu or used as a shortcut. This is done by attaching metadata to the command, as displayed in the example.

```
public void Initialize(IUnityContainer unityContainer)
{
  var uiDefManager = unityContainer.Resolve<UIDefinitionManager>();

  //Creates a collection of ui element definitions.
  var uiElementCollection = new UIDefinitionCollection()
  {
      new DialogDefinition<ICustomDialogViewModel, CustomDialogViewModel, ICustomDialogView, Cu
  };
  // We register the elements of the collection with the UIDefinitionManager
  uiElementCollection.RegisterAll(uiDefManager);

  // We create the command collection and add the command definitions.
  var cmdCollection = new CommandDefinitionCollection<IPluginCommands>()
  {
     {
        c => c.ContextCommand,
        new ManagedCommand<IBitmap>()
        {
           ExecuteHandler = o=>
           {
           },
           Metadata = new MetadataCollection()
           {
              new ContextMenuOption()
              {
                  MenuPath = new MenuPath() { "Here" },
                  PanelViewModel = typeof(ITreeViewModel<>)
              },
           new AutoInvalidateOnSelectionInterface<ISelectedSolution>()
           }
        }
     },
     {
        c=>c.ShowDialogCommand,
        new ManagedCommand()
        {
          ExecuteHandler = (o)=>
          {
            var dlgSrv = unityContainer.Resolve<IDialogSvc>();
            var dialogVM = unityContainer.Resolve<ICustomDialogViewModel>();
```

```
            dialogVM.Message = "Custom Dialog";
            if (dlgSrv.ShowDialog(dialogVM) == true)
            {
              dlgSrv.ShowMessageBox(dialogVM.Message);
            }
          },
          Metadata = new MetadataCollection()
          {
            new MainMenuOption()
            {
              MenuPath = new MenuPath() { MenuConstants.File_2, "My Menu Option", 350 },
            },
            new ToolTip("Invokes a custom dialog from a plugin"),
            new KeyShortcut() { Key = Key.B, Modifiers= ModifierKeys.Control | ModifierKeys.Shi
          }
        }
    }
  };
  cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
 }
```

The dialog window

This custom dialog will contain a button that will exacute a command to display a messagebox.

```
this.ShowMessageBox = new ManagedCommand()
{
  ExecuteHandler = o =>
  container.Resolve<IDialogSvc>().ShowMessageBox
  (
  "Message box from a plugin",
  "Message box title",
  MessageBoxButton.OK,
  MessageBoxImage.Exclamation
  )
 };
```

The full code for the example:

CustomUIConfigPlugin.cs

```
 using System;
 using System.Collections.Generic;
 using System.Linq;
 using System.Text;
 using FTC.SDKInterface.Plugins;
 using FTC.SDKInterface.Selections;
 using FTC.SDKInterface.Services;
 using Feat.UIComposition;
 using Feat.UIComposition.Common;
 using Feat.UIComposition.Config;
 using Feat.UIComposition.Paneling;
```

```
using FTC.SDKInterface.Constants;
using UIPlugins.PluginWithMenuAndDialog.CustomDialog;
using Feat.UIComposition.Dialogs;
using Feat.UIComposition.Commanding;
using Feat.UIComposition.Menus;
using Feat.UIComposition.Shortcuts;
using System.Windows.Input;
using Feat.CAL.Services;
using Feat.UIComposition.ContextMenus;
using Feat.Core;
using FTC.SDKInterface.Solution;

namespace UIPlugins.PluginWithMenuAndDialog
{
   /// <summary>
   /// This is an example of a plugin that registers a command and displays a custom dialog ar
   /// </summary>
   public class PluginWithMenuAndDialogPlugin : IPluginService
   {
      public string Description
      {
         get { return "A plugin with a menu item and dialog"; }
      }
      public string Name
      {
         get { return this.GetType().Name; }
      }

      public string Version
      {
         get { return "1.1.1"; }
      }

      public IEnumerable<int> FeatureIds
      {
         get { return null; }
      }

      /// <summary>
      /// The initialization method has the role of registering with Scene composer different
      /// </summary>
      /// In this example the initliaze method will register a menu command and a custom dialo
      ///
      /// Registering Commands:
      /// To register a command first an interface that derives from ICommandContainer must be
      /// The interface will contain properties for each of the commands that will be register
      /// The interface will be implemented internally by Scene Composer.
      /// The plugin can then register the commands the UICommandManager service. This can be
      /// directly or using the helper class CommandDefinitionCollection that simplifies the r
      /// commands. \n
      ///
      /// The command can be invoked from code, (see other more complex examples for a demonst
      /// or can be displayed in the menu or used as a shortcut. This is done by attaching met
```

```
/// as displayed in the example.
/// <param name="unityContainer"></param>
public void Initialize(IUnityContainer unityContainer)
{
   var uiDefManager = unityContainer.Resolve<UIDefinitionManager>();

   //Creates a collection of ui element definitions.
   // In this case we just register a dialog.
   // A ui element has 4 elements :
   // - a view model interface
   // - a view model class implementation
   // - a view interface
   // - a view class implementation
   var uiElementCollection = new UIDefinitionCollection()
   {
      new DialogDefinition<ICustomDialogViewModel, CustomDialogViewModel, ICustomDialog\
   };
   // We register the elements of the collection with the UIDefinitionManager
   uiElementCollection.RegisterAll(uiDefManager);

   // We create the command collection and add the command definitions.
   var cmdCollection = new CommandDefinitionCollection<IPluginCommands>()
   {
      {
         c => c.ContextCommand,
         new ManagedCommand<IBitmap>()
         {
            ExecuteHandler = o=>
            {
               //unityContainer.Resolve<FTC.SDKInterface.Selections.ISelectedScene>().Va
            },
            Metadata = new MetadataCollection()
            {
               new ContextMenuOption()
               {
                  MenuPath = new MenuPath() { "Here" },
                  PanelViewModel = typeof(ITreeViewModel<>)
               },
               new AutoInvalidateOnSelectionInterface<ISelectedSolution>()
            }

         }
      },
      {
         // We specify the command for which we are registering an implementation,
         // The type of c will be inferred to be IPluginCommands, so we don't have to sp
         c=>c.ShowDialogCommand,
         //
         new ManagedCommand()
         {
            ExecuteHandler = (o)=>
            {
               var dlgSrv = unityContainer.Resolve<IDialogSvc>();
```

```
                    // Invokes a dialog that was previously registered with the UIDefinitionM
                    // We first create the viewmodel for the dialog
                    var dialogVM = unityContainer.Resolve<ICustomDialogViewModel>();
                    // We configure the view model properties
                    dialogVM.Message = "Custom Dialog";

                    // We show the dialog. The dialog window will be constructed based on the
                    // UIDefinitionManager service.
                    if (dlgSrv.ShowDialog(dialogVM) == true)
                    {
                       dlgSrv.ShowMessageBox(dialogVM.Message);
                    }
                 },
                 Metadata = new MetadataCollection()
                 {
                    // The main menu option to be created.
                    // The menu path is specifies the location where the menu should be displ
                    // The menu constant class contains the main menu definitions used by sce
                    // The plugin can an other main menu options too.
                    // The tokens of the path can be specified either with commas, or using m
                    // The each token of a path can be followed by an integer that represents
                    // in the menu. The option present in the scene composer menu, have order
                    // so plugins can add items in between the existing items
                    // A menu can be devided in several groups, in the example below, the ite
                    new MainMenuOption()
                    {
                       MenuPath = new MenuPath() { MenuConstants.File_2, "My Menu Option", 35
                    },
                    // Specifies a tooltip for the command, used for the menu items and the k
                    new ToolTip("Invokes a custom dialog from a plugin"),
                    //Specifies a shortcut key for the command.
                    new KeyShortcut() { Key = Key.B, Modifiers= ModifierKeys.Control | Modifi
                 }
              }
           }
        };
        // We register the commands.
        cmdCollection.RegisterAll(unityContainer.Resolve<UICommandManager>());
     }
  }

  /// <summary>
  /// The interface that contains the command definitions.
  /// </summary>
  /// The main role of this interface is to uniquely identify the command and to provide an e
  /// No implementation of this interface is required, Scene COmposer will generate a concret
  public interface IPluginCommands : ICommandContainer
  {
     ManagedCommand ShowDialogCommand { get; }
     ManagedCommand<IBitmap> ContextCommand { get; }
  }
}
```

ICustomDialogViewModel.cs

```
using System;
using System.Collections.Generic;
using Feat.Core;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
   /// <summary>
   /// The dialog view model interface. This interface can be used to exchange information bet
   /// </summary>
   public interface ICustomDialogViewModel
   {
      string Message { get; set; }

   }
}
```

ICustomDialogView.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Feat.Core;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
   /// <summary>
   /// The interface for the dialog view. It contains no methods as the view does not communic
   /// </summary>
   public interface ICustomDialogView : IDialogView
   {
   }
}
```

CustomDialogViewModel.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Windows;
using FTC.SDKInterface.Services;
using Feat.Core;
using System.Windows.Input;
using Feat.UIComposition.Commanding;
using Feat.CAL.Services;

namespace UIPlugins.PluginWithMenuAndDialog.CustomDialog
{
```

```
    class CustomDialogViewModel : ICustomDialogViewModel
    {
       public CustomDialogViewModel(IUnityContainer container)
       {
          // We create a simple command to demo using a Scene Composer message box
          // This is the way to show message box so that it will conform to the Scene Composer
          this.ShowMessageBox = new ManagedCommand()
          {
             ExecuteHandler = o =>
                container.Resolve<IDialogSvc>().ShowMessageBox
                (
                   "Message box from a plugin",
                   "Message box title",
                   MessageBoxButton.OK,
                   MessageBoxImage.Exclamation
                )
          };
       }

       /// <summary>
       /// The command for showing a dialog box, that will be bound to the view.
       /// </summary>
       public ICommand ShowMessageBox { get; set; }

       /// <summary>
       /// The message that will be retuned by the viewmodel.
       /// </summary>
       public string Message { get; set; }
    }
 }
```

CustomDialogView.xaml

```
 <!--
     The actual UI for the view. Most of the look and feel for the application is inherited by t
     The dialog window must however specify the Style property to ensure consistent look and fee
     The DialogWindow style resource is defined within Scene Composer
  -->

  <ui:DialogWindow
      x:Class="UIPlugins.PluginWithMenuAndDialog.CustomDialog.CustomDialogView"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:ui="clr-namespace:Feat.Core;assembly=Feat.Core"
      Title="Plugin dialog"
      Style="{DynamicResource DialogWindow}" >
     <StackPanel Margin="15">
        <Label Content="This is a custom dialog defined in a plugin"></Label>
        <StackPanel>
           <Label Content="Message:"></Label>
           <!--We use databiding to edit the MessageProperty-->
           <TextBox Text="{Binding Message}"></TextBox>
        </StackPanel>
        <!--We  use databinding to assign the ShowMessageBox command to the button-->
```

```
        <Button Command="{Binding ShowMessageBox}" Content="Show message Box" Margin="0,30,0,10"
        <StackPanel Orientation="Horizontal" Margin="10">
           <Button Click="OnOk" Margin="10" Content="OK"></Button>
           <Button Click="OnCancel" Margin="10" Content="Cancel"></Button>
        </StackPanel>
     </StackPanel>
  </ui:DialogWindow>
```

Revision #1
Created 2023-02-15 07:54:38 UTC by Tsuyoshi.Kato
Updated 2023-02-15 07:55:48 UTC by Tsuyoshi.Kato