DevLynx Blog - CAB

Echoes from the Cave

Oct 9th 2007

CAB Test Application

At my day job, we are beginning to look at the next generation of our product line. The current generation is written in Borland Delphi and is starting to show it's age. Over the years it has become tightly coupled and is increasingly difficult to maintain. I am currently investigating which technologies to use for our rewrite. We have already decided to develop in C#. Anything and everything else is still to be determined.

A couple of months ago, I discovered Microsoft Composite Application Block (CAB). CAB was written by Microsoft's patterns & practices team as a framework for decoupling large applications (http://www.cabpedia.com).

To test the capabilities of CAB I have decided to write an Address Book as a test application. Furthermore, I am going to attempt to blog as a means of documenting my thoughts and progress. This will be my first foray into the blogshere.

My next entry will discuss my goals followed by a brief set of informal requirements for the test application.

dlx

Oct 10th, 2007

CAB Test App Goals

Users expect WinForms applications to provide a rich user experience (Ux). One of the main goals of the test app is to determine if the CAB can support this type of Ux. The simple nature of the test app does not have the functionality to necessitate all of the controls with which I want to experiment; therefore, some of the Ux will be a bit contrived.

A second major goal will be to experiment with Microsoft Enterprise Library.

I will be using controls from Developer Express (Dx) which provide an excellent collection of WinForms controls. In addition, I will use the CAB DevExpress Extension Kit which provides pre-built support for Dx controls within the CAB.

Ux Goals:

1. Switch between a menu/toolbar and a Ribbon control in the same application. This is more of a curiosity for me. Can navigation be abstracted enough so that it does not matter what type of controls are used?
2. Nothing on the shell form should be pre-populated. It should consist of a menu/toolbar or a Ribbon control and a statusbar.
3. The user should be able to use super tooltips or normal tooltips. When a command is disabled the super tooltip information should change to inform the user how the command can be enabled.
4. The navigation system should be able to include button/menu item, checkbox, dropdown buttons (sometimes called split buttons), galleries and button groups for the Ribbon, as well as other controls (e.g. font name combobox).
5. Dx XtraBars includes a docking system. I need to determine how well they can be used as Workspaces. The dockable panels need to be created on the fly and properly placed.
6. Buttons can handle two commands simultaneously. Need to investigate how well this works.
7. Navigation can also be done with the so called Outlook bar. Need to experiment with this type of navigation.
8. For long processes the user expects the application to show progress. Will need to add a progress bar to the statusbar on the fly and then remove it when it is no longer needed.
9. Since the CAB only adds items to the right and bottom of UIElements we need to have some method to create them in a more specific order regardless of the module load order.
10. Mapping: I want to embed a mapping tool into the test app (it is an address book after all). See what it would take to use either Microsoft Live Search Maps or Google Maps.

Other Goals:

1. Decoupled: this is, of course, redundant; the CAB is all about decoupling.
2. Globalization: all strings, images and other Ux elements must come from a resource file.

3. Security: CAB can restrict module loading via role based security. So some parts of the test application must be secured. Look into also using the Encryption Application Block to provide additional functionality.
4. Logging: the test app should log significant events. Use the Logging Application Block to provide this functionality.
5. Configuration: the test app should not access the registry. The app should provide for shared as well as user configuration files. Use the Configuration Application Block to provide this functionality.
6. Database agnostic: Use Dx eXpress Persistent Objects (XPO) an object-relational mapping (ORM) tool. I'm not sure that we will use an ORM for our big rewrite at my day job, but I want to experiment with it here.

dlx

Oct 13, 2007

CAB Test App - Features

I will be designing and implementing an address book as a test app. I chose an address book for a very pragmatic reason. My sister was visiting a couple of weeks ago and was searching for a simple address book app to organize her personal contacts; I was searching for a small project to test the CAB. It didn't take a bolt of lightning for us to realize the mutual benefits.

Required Components:

1. Name will have various components including: Display Name, First Name, Last Name, Title, Suffix, Nickname
2. Address will include the usual items: Street (multiline), City, State/Province, Postal Code, Country
   a. Multiple names can be associated with a single address.
   b. Multiple addresses can be associated with a single person - (home, work, summer house, etc.) This is not actually a required feature but it will give me a chance to delve into the more sophisticated aspects of Dx eXpress Persistent Objects (XPO). This feature will test adding the address type into the many-many relationship linking object in XPO.
3. Phone number will include the number and what type of phone (home, cell, work, fax, etc.) A phone number can be associated with a name or an address.
4. Email address and type (personal, work, etc.)
5. Website and type (personal, work, etc.)
6. Notes associated with the name or address
7. Tags - tags provide a way to filter names. The user will be able to create tags such as Friends, Family, Business, Xmas Card, etc. Any number of tags can be associated with a name/address pair.
8. Reporting and Labels - need to support a variety of labels for printing as well as data reporting.
   a. I will be using Dx XtraReports for reporting.
   b. All reports will be stored externally and loaded automatically by the app. This allows me to create a new report and give it to users (OK, my sister) without recompiling the app.
   c. Users must be able to create custom reports which are indistinguishable from reports that are shipped with the app. This allows users to share their reports.
9. Mapping - Addresses must be able to be mapped using Google Maps or Microsoft Live Search Maps.

Nice-to-have Features:

1. Important dates associated with names (Birthday, Anniversary, etc.)
2. Other information associated with names (Gender, Picture, Avitar, etc.)
3. Name styles - depending on the occasion, we want to print "Dr. John Smith III" on an address label and other times we want to print "Johnny Smith". Supplying a method to enter this information and select the name style at print time would be useful.

4. Use the USPS web services to create standardized addresses and get Zip Code information
5. vCard support
6. Be able to backup the data
7. Tickler dates and times - this would allow me to play with a web service that stores the information and sends email at the appropriate time.
8. Share and merge data from multiple address books via a web service. With this feature, I can let my sister maintain family and mutual friends addresses and I reap the benefits (bwah-ha-ha!!).
    a. Use a shared flag for each name so that each name can be either shared or private. (added 16 Jan 2008).

Since this is a test app (and a fairly simple one) this list will be the sum total of my requirements document.

No doubt other features will creep into the app to fulfill all of the goals that I have for this test. If a new feature is significant I will address it in a future blog.

dlx

Oct 23, 2007

Starting the CAB Test App

I started this blog to document what I am doing while creating a CAB project. This documentation accomplishes a number of things: recording my thoughts and implementation enables me to solidify the process in my own mind. It also provides bread crumbs for creating the larger project at the day job. And finally, it allows others who are experimenting with the CAB to gain some benefit from my thought process.

These entries will become a "how to" on CAB. There will be a lot of code snippets and discussions of implementation. So unless you have "embraced your inner geek" (or you are one of my family members and obligated to read this) now is the time to bail out ;-)

At the day job we have four or five applications which could take advantage of the CAB. So my first (erroneous) thought was that there would be four or five shell applications generated by the SCSF - one for each product. The same basic code would be present in all of these shells, a direct violation of "Don't Repeat Yourself" (DRY). What really needs to be done is to create a product independent shell; all applications will use this common shell.

NOTE: This is a test app - I am not (yet) a CAB expert and have not written any CAB apps beyond creating a "Hello World" example. If I had enough experience with the CAB...well, I wouldn't be writing this test app. So what you see here will almost certainly change throughout the development process (hey, just like a normal project). I'll add and backtrack as I learn new things or program myself into a corner. So, if you see me doing something wrong (or just plain stupid), or you see a better way, please let me know. After the next entry I'll start posting full source code so that you can see the entire picture.

NOTE 2: I am assuming that you know the basics of both Visual Studio, Developer Express controls, and have worked through at least the SCSF Hands On Labs.

Our first task is to create the product independent shell application. We will make code modifications to use the Developer Express controls and allow the user to select either a standard menu or the Office 2007 Ribbon control.

Off to work: create a new Smart Client Solution, uncheck "Create a separate module to define the layout for the shell". Since this is product independent we will take a minimalist approach for the shell; another, product specific, module will define the layout.

In the Infrastructure.Library project, add a reference to CABDevExpress. In the SmartClientApplication.cs change the FormShellApplication to an XtraFormApplication.

In ShellForm.cs change the form to descend from the Dx XtraForm. In the ShellForm (design mode) 'cry havoc' and delete all controls. Add a Dx BarManager, DefaultLookAndFeel and dock a DeckWorkspace to fill the rest of the form. In the Dx toolbar delete the default toolbar. Since we want this to work for all applications the form should define only the minimal layout. Toolbars, commands and status bar items will all be created by application specific modules.

Add the appropriate references and using statements. Rename components to something more descriptive...in other words, cleanup work. In ShellForm.cs we will need to remove the references to the status bar label and the exit menu item.

After cleanup, my resulting ShellForm.cs is:

```
namespace DevLynx.ShellApp.Infrastructure.Shell
{
    using DevLynx.ShellApp.Infrastructure.Interface.Constants;
    using DevExpress.XtraEditors;
    using DevExpress.XtraBars;

    /// <summary>
    /// Main application shell view.
    /// </summary>
    public partial class ShellForm : XtraForm, ISmartPartInfoProvider
    {
        private Microsoft.Practices.CompositeUI.SmartParts.
                SmartPartInfoProvider infoProvider;

        /// <summary>
        /// Default class initializer.
        /// </summary>
        public ShellForm()
        {
            InitializeComponent();
            this.infoProvider = new Microsoft.Practices.CompositeUI.
                  SmartParts.SmartPartInfoProvider();
            layoutWorkspace.Name = WorkspaceNames.LayoutWorkspace;
        }

        internal Bar MainMenu
        {
            get { return mainMenu; }
        }

        internal Bar MainStatus
        {
            get { return statusBar; }
        }

        internal Bars MainToolbar
        {
            get { return barManager.Bars; }
        }

        public Microsoft.Practices.CompositeUI.SmartParts.ISmartPartInfo
              GetSmartPartInfo(Type smartPartInfoType)
        {
            Microsoft.Practices.CompositeUI.SmartParts.
                  ISmartPartInfoProvider ensureProvider = this;
            return this.infoProvider.GetSmartPartInfo(smartPartInfoType);
        }
    }
```

```
}
```

Next we need to make modifications to show either a Ribbon form or a menu/toolbar form. Our first step in this feature is to create a Dx Ribbon form. In the design mode of the Ribbon form, remove the default ribbon page - we want as clean of a form as possible. Add a DefaultLookAndFeel, an ApplicationMenu and dock a DeckWorkspace to fill the rest of the form. Do the same type of cleanup as in the menu based form.

We also want to expose the Ribbon, status bar and application menu so that UI items can be added.

My resulting ShellRibbonForm is:

```
namespace DevLynx.ShellApp.Infrastructure.Shell
{
    using System;
    using DevExpress.XtraBars;
    using Microsoft.Practices.CompositeUI.SmartParts;
    using DevExpress.XtraBars.Ribbon;
    using Microsoft.Practices.CompositeUI.EventBroker;
    using DevLynx.ShellApp.Infrastructure.Interface;
    using DevLynx.ShellApp.Infrastructure.Interface.Constants;

    public partial class ShellRibbonForm :
          DevExpress.XtraBars.Ribbon.RibbonForm, ISmartPartInfoProvider
    {
        private Microsoft.Practices.CompositeUI.SmartParts.
            SmartPartInfoProvider infoProvider;

        public ShellRibbonForm()
        {
            InitializeComponent();
            this.infoProvider = new Microsoft.Practices.CompositeUI.
                  SmartParts.SmartPartInfoProvider();
            layoutWorkspace.Name = WorkspaceNames.LayoutWorkspace;
        }

        internal RibbonControl MainMenu
        {
            get { return ribbon; }
        }

        internal RibbonStatusBar MainStatus
        {
            get { return ribbonStatusBar; }
        }

        internal ApplicationMenu ApplicationMenu
        {
            get { return applicationMenu; }
        }

        public Microsoft.Practices.CompositeUI.SmartParts.ISmartPartInfo
              GetSmartPartInfo(Type smartPartInfoType)
        {
            Microsoft.Practices.CompositeUI.SmartParts.
                  ISmartPartInfoProvider ensureProvider = this;
            return this.infoProvider.GetSmartPartInfo(smartPartInfoType);
        }
    }
```

```
}
```

We have now created the UI elements but we cannot yet choose between the two forms. Here we turn to the ShellApplication class. ShellApplication currently contains the Main method entry point which instantiates the ShellForm via the ShellApplication class. So we want to extract a class which just contains the Main method entry point. The new ShellMain class will call either the ShellApplication (standard menu) or it will call the ShellRibbonApplication (Ribbon control). The ShellMain class will reside in the Shell project.

My ShellMain.cs class:

```
namespace DevLynx.ShellApp.Infrastructure.Shell
{
    using System;
    using System.Configuration;
    using System.Collections.Specialized;
    using System.Windows.Forms;
    using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;
    using DevExpress.XtraEditors;

    /// <summary>
    /// Main application entry point class.
    /// </summary>
    class ShellMain
    {
        /// <summary>
        /// Application entry point.
        /// </summary>
        [STAThread]
        static void Main()
        {
            // grab the setting for which form to show from AppSetting
            // in the configuration file.
            NameValueCollection appSettings =
                  ConfigurationManager.AppSettings;
            string IsRibbonForm = appSettings.Get("IsRibbonForm");
            bool isRibbon = Convert.ToBoolean(IsRibbonForm);
            bool enableFormSkins = false;

            if (!isRibbon)
                // Don't use the bonus skins for Ribbons because I don't think
                // they follow the Microsoft Ribbon Guidelines closely enough.
                DevExpress.UserSkins.BonusSkins.Register();

            DevExpress.UserSkins.OfficeSkins.Register();
            DevExpress.Skins.SkinManager.Default.
                  RegisterAssembly(typeof(
                  DevExpress.UserSkins.Office2007Bonus).Assembly);
            if (enableFormSkins)
                DevExpress.Skins.SkinManager.EnableFormSkins();

#if (DEBUG)
            if (isRibbon)
                ShellRibbonApplication.RunInDebugMode();
            else
                ShellApplication.RunInDebugMode();
#else // (RELEASE)
            if (isRibbon)
                ShellRibbonApplication.RunInReleaseMode();
            else
                ShellApplication.RunInReleaseMode();
```

```
#endif // if (DEBUG)
        }
    }
}
```

The modified ShellApplication class controls the standard menu form but without the Main method entry point. My modified ShellApplication.cs class is:

```
namespace DevLynx.ShellApp.Infrastructure.Shell
{
    using System;
    using DevLynx.ShellApp.Infrastructure.Library;
    using Microsoft.Practices.CompositeUI;
    using DevLynx.ShellApp.Infrastructure.Interface.Constants;
    using CABDevExpress.UIElements;
    using System.Windows.Forms;
    using DevExpress.XtraEditors;
    using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;

    class ShellApplication : SmartClientApplication<WorkItem, ShellForm>
    {
        /// <summary>
        /// Sets the extension site registration after the shell has been created.
        /// </summary>
        protected override void AfterShellCreated()
        {
            base.AfterShellCreated();

            RootWorkItem.UIExtensionSites.RegisterSite(
                 UIExtensionSiteNames.MainMenu, this.Shell.MainMenu);
            RootWorkItem.UIExtensionSites.RegisterSite(
                 UIExtensionSiteNames.MainStatus, this.Shell.MainStatus);
            RootWorkItem.UIExtensionSites.RegisterSite(
                   UIExtensionSiteNames.MainToolbar,
                   new BarsUIAdapter(this.Shell.MainToolbar));
        }

        internal static void RunInDebugMode()
        {
            Application.SetCompatibleTextRenderingDefault(false);
            new ShellApplication().Run();
        }

        internal static void RunInReleaseMode()
        {
            AppDomain.CurrentDomain.UnhandledException +=
                  new UnhandledExceptionEventHandler(AppDomainUnhandledException);
            Application.SetCompatibleTextRenderingDefault(false);
            try
            {
                new ShellApplication().Run();
            }
            catch (Exception ex)
            {
                HandleException(ex);
            }
        }

        private static void AppDomainUnhandledException(
                 object sender, UnhandledExceptionEventArgs e)
        {
            HandleException(e.ExceptionObject as Exception);
        }
```

```
        private static void HandleException(Exception ex)
        {
            if (ex == null)
                return;

            ExceptionPolicy.HandleException(ex, "Default Policy");
            XtraMessageBox.Show("An unhandled exception occurred,
                                and the application is terminating.
                                For more information, see your
                                Application event log.");
            Application.Exit();
        }
    }
}
```

We now need a corresponding class to handle the Ribbon control. Create a new ShellRibbonApplication.cs to instantiate the ShellRibbonForm. My ShellRibbonApplication.cs:

```
namespace DevLynx.ShellApp.Infrastructure.Shell
{
    using System;
    using System.Windows.Forms;
    using Microsoft.Practices.CompositeUI;
    using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;
    using DevLynx.ShellApp.Infrastructure.Library;
    using DevLynx.ShellApp.Infrastructure.Interface.Constants;
    using DevExpress.XtraEditors;

    internal class ShellRibbonApplication :
              SmartClientApplication<WorkItem, ShellRibbonForm>
    {
        internal static void RunInDebugMode()
        {
            Application.SetCompatibleTextRenderingDefault(false);
            ShellRibbonApplication app = new ShellRibbonApplication();
            app.Run();
        }

        internal static void RunInReleaseMode()
        {
            AppDomain.CurrentDomain.UnhandledException +=
                   new UnhandledExceptionEventHandler(AppDomainUnhandledException);
            Application.SetCompatibleTextRenderingDefault(false);
            try
            {
                ShellRibbonApplication app = new ShellRibbonApplication();
                app.Run();
            }
            catch (Exception ex)
            {
                HandleException(ex);
            }
        }

        protected override void BeforeShellCreated()
        {
            base.BeforeShellCreated();
        }

        /// <summary>
        /// Sets the extension site registration after the shell has been created.
        /// </summary>
        protected override void AfterShellCreated()
        {
```

```
            base.AfterShellCreated();

            RootWorkItem.UIExtensionSites.RegisterSite(
                    UIExtensionSiteNames.MainMenu, this.Shell.MainMenu);
            RootWorkItem.UIExtensionSites.RegisterSite(
                    UIExtensionSiteNames.MainStatus, this.Shell.MainStatus);
            //Unfortunately we don't yet have a UIExtensionSite for an
            //ApplicationMenu - we will create one of these later
            //RootWorkItem.UIExtensionSites.RegisterSite(
            //      UIExtensionSiteNames.ApplicationMenu,
            //      this.Shell.ApplicationMenu);
        }

        private static void AppDomainUnhandledException(
                 object sender, UnhandledExceptionEventArgs e)
        {
            HandleException(e.ExceptionObject as Exception);
        }

        private static void HandleException(Exception ex)
        {
            if (ex == null)
                return;

            ExceptionPolicy.HandleException(ex, "Default Policy");
            XtraMessageBox.Show("An unhandled exception occurred,
                                 and the application is terminating.
                                 For more information, see your
                                 Application event log.");
            Application.Exit();
        }
    }
}
```

Obviously when you compile you will need to resolve all of the little changes that were not described here as well as adding references to the Dx specific modules. In the next entry I will supply source code so that you can see the full code in context.

We have created the foundation for a general purpose shell application which supports either a standard menu or a Ribbon control. In the next entry we will implement functionality to add buttons/menu items to either of the two forms.

dlx

Oct 28, 2007

Abstracting CAB Commands with Developer Express Controls

In order to support either the Ribbon control or the menu/toolbar paradigm we must abstract the creation of UI extensions. When using menu items or buttons, either on the Ribbon or toolbar, Developer Express makes this easy. Both menu items and buttons use a BarButtonItem. This allows us to abstract where the instance of BarButtonItem is placed.

To facilitate the abstraction I have created a UxExtension service that any CAB WorkItem can call to add a BarButtonItem. The BarButtonItem creation is described by the UxExtensionEventArgs class. This class will eventually contain members that will describe all of the BarButtonItem properties so that a rich user experience can be obtained. However, the first iteration just provides enough information to get the button into the user interface.

The UxExtensionEventArgs:

```
namespace DevLynx.ShellApp.Infrastructure.Interface
{
    using System;
    using System.Drawing;

    /// <summary>
    /// Describes a button or menu item that will be added
    /// to the user interface. UxExtensionEventArgs abstracts
    /// the information enough so that we can diplay either a
    /// menu/toolbar or Ribbon paradigm to the user. This is
    /// possible because Developer Express uses a common object
    /// (<see cref="DevExpress.XtraBars.BarButtonItem"/>) for
    /// it's menus, toolbars and the Ribbon control.
    /// Thank you Deverloper Express!
    /// </summary>
    public class UxExtensionEventArgs : EventArgs
    {
        /// <summary>
        /// Initializes a new instance of the
        /// <see cref="UxExtensionEventArgs"/> class.
        /// </summary>
        public UxExtensionEventArgs()
        {
            CommandRank = CommandRank.Default;
            Invoker = "ItemClick";
        }

        private string commandName;
        /// <summary>
        /// Gets or sets the name of the CAB command.
        /// </summary>
        /// <value>The name of the CAB command.</value>
        public string CommandName
        {
            get { return commandName; }
            set { commandName = value; }
        }
```

```
private string invoker;
/// <summary>
/// Gets or sets the command invoker which is the event
/// name that is fired when the command is invoked.
/// </summary>
/// <value>The name of the invoker event.</value>
public string Invoker
{
    get { return invoker; }
    set { invoker = value; }
}

private string text;
/// <summary>
/// Gets or sets the text displayed with the button.
/// </summary>
/// <value>The text.</value>
public string Text
{
    get { return text; }
    set { text = value;}
}

private Image smallImage;
/// <summary>
/// Gets or sets the small image displayed on a
/// smaller button.
/// </summary>
/// <value>The image.</value>
public Image SmallImage
{
    get { return smallImage; }
    set { smallImage = value; }
}

private Image largeImage;
/// <summary>
/// Gets or sets the large image displayed on a
/// larger button.
/// </summary>
/// <value>The image.</value>
public Image LargeImage
{
    get { return largeImage; }
    set { largeImage = value; }
}

private CommandRank commandRank;
/// <summary>
/// Gets or sets the <see cref="CommandRank"/> which decides
/// how the command is displayed on the ribbon or menu/toolbar.
/// </summary>
/// <value>The command rank.</value>
public CommandRank CommandRank
{
    get { return commandRank; }
    set { commandRank = value; }
}
```

```
        private string page;
        /// <summary>
        /// Gets or sets the ribbon page where this command is to
        /// be placed.
        /// </summary>
        /// <value>The ribbon page.</value>
        public string Page
        {
            get { return page; }
            set { page = value; }
        }

        private string group;
        /// <summary>
        /// Gets or sets the ribbon group where this command is to
        /// be placed.
        /// </summary>
        /// <value>The ribbon group.</value>
        public string Group
        {
            get { return group; }
            set { group = value; }
        }

        private string menu;
        /// <summary>
        /// Gets or sets the menu where this command is to be
        /// placed.
        /// </summary>
        /// <value>The menu.</value>
        public string Menu
        {
            get { return menu; }
            set { menu = value; }
        }

        private string toolbar;
        /// <summary>
        /// Gets or sets the toolbar where this command is
        /// to be placed. If Toolbar is blank the command
        /// is not placed on any toolbar.
        /// </summary>
        /// <value>The toolbar.</value>
        public string Toolbar
        {
            get { return toolbar; }
            set { toolbar = value; }
        }
    }
}
```

The UxExtension service currently provides a single method to add a command. The AddCommand method takes the WorkItem that is adding the command and a UxExtensionEventArgs as parameters. The AddCommand determines which paradigm to use and calls either the AddDxRibbonCommand or the AddDxMenuCommand. The

UxExtension service will eventually have methods to add a top level menu, a toolbar and Ribbon pages and groups. The current UxExtension class:

```
namespace DevLynx.ShellApp.Infrastructure.Library.Services
{
    using System;
    using Microsoft.Practices.CompositeUI;
    using DevExpress.XtraBars;
    using DevExpress.XtraBars.Ribbon;
    using DevLynx.ShellApp.Infrastructure.Interface.Constants;
    using DevLynx.ShellApp.Infrastructure.Interface;
    using DevLynx.ShellApp.Infrastructure.Interface.Services;

    /// <summary>
    /// Implementation of the IUxExtension service interface.
    /// </summary>
    [Service(typeof(IUxExtension))]
    public class UxExtension : IUxExtension
    {
        private WorkItem rootWorkItem;
        private bool? formHasRibbonControl;

        /// <summary>
        /// Initializes a new instance of the <see cref="UxExtension"/>
        /// class.
        /// </summary>
        /// <param name="rootWorkItem">The root work item.</param>
        public UxExtension([ServiceDependency] WorkItem rootWorkItem)
        {
            this.rootWorkItem = rootWorkItem;
            if (formHasRibbonControl == null)
            {
                IGlobalInformation globalInformation =
                    rootWorkItem.Services.Get<IGlobalInformation>(true);
                formHasRibbonControl =
                    globalInformation.FormHasRibbonControl;
            }
        }

        /// <summary>
        /// Adds a CAB command to the user interface.
        /// </summary>
        /// <param name="workItem">The work item that is adding this
        /// command.</param>
        /// <param name="eventArgs">The <see cref="UxExtensionEventArgs"/>
        /// instance containing the command data.</param>
        /// <returns>
        /// The <see cref="DevExpress.XtraBars.BarButtonItem"/>
        /// that was added.
        /// </returns>
        public BarButtonItem AddCommand(WorkItem workItem,
            UxExtensionEventArgs eventArgs)
        {
            if ((bool)formHasRibbonControl)
            {
                return AddDxRibbonCommand(workItem, eventArgs);
            }
            else
            {
```

```
            return AddDxMenuCommand(workItem, eventArgs);
        }
    }

    private BarButtonItem AddDxRibbonCommand(WorkItem workItem,
        UxExtensionEventArgs eventArgs)
    {
        AddRibbonInfrastructureIfNecessary(eventArgs.Page,
            eventArgs.Group);
        BarButtonItem button = InitializeButton(eventArgs);
        button.RibbonStyle = GetRibbonStyle(eventArgs.CommandRank);
        workItem.UIExtensionSites[GetRibbonGroupName(eventArgs.Page,
            eventArgs.Group)].Add<BarButtonItem>(button);
        workItem.Commands[eventArgs.CommandName].AddInvoker(button,
            eventArgs.Invoker);
        return button;
    }

    private BarButtonItem AddDxMenuCommand(WorkItem workItem,
        UxExtensionEventArgs eventArgs)
    {
        AddMenuInfrastructureIfNecessary(eventArgs.Menu,
            eventArgs.Toolbar);
        BarButtonItem button = InitializeButton(eventArgs);
        workItem.UIExtensionSites[GetMainMenuName(eventArgs.Menu)].
            Add<BarButtonItem>(button);
        workItem.Commands[eventArgs.CommandName].AddInvoker(
            button, eventArgs.Invoker);

        if (!String.IsNullOrEmpty(eventArgs.Toolbar))
        {
            workItem.UIExtensionSites[GetToolbarName(
                eventArgs.Toolbar)].Add<BarButtonItem>(button);
        }
        return button;
    }

    /// <summary>
    /// Adds the menu/toolbar infrastructure if necessary.
    /// Menus and toolbars should be created prior to adding
    /// commands since more settings can be initialized. This
    /// is a last resort which will ensure that the menu and
    /// toolbars are created with default parameters if they
    /// do not already exist.
    /// </summary>
    /// <param name="menu">The menu in which this command will
    /// be added.</param>
    /// <param name="toolbar">The toolbar on which this command
    /// will be displayed.</param>
    private void AddMenuInfrastructureIfNecessary(string menu,
        string toolbar)
    {
        string mainMenuName = GetMainMenuName(menu);
        if (!rootWorkItem.UIExtensionSites.Contains(mainMenuName))
        {
            BarSubItem mainMenuItem = new BarSubItem();
            mainMenuItem.Caption = menu;
            rootWorkItem.UIExtensionSites[
```

```
                UIExtensionSiteNames.MainMenu].
                Add<BarSubItem>(mainMenuItem);
            rootWorkItem.UIExtensionSites.RegisterSite(
                mainMenuName, mainMenuItem);
        }
        string toolbarName = GetToolbarName(toolbar);
        if (!String.IsNullOrEmpty(toolbar)
            && !rootWorkItem.UIExtensionSites.Contains(toolbarName))
        {
            Bar toolbarItem = new Bar();
            toolbarItem.Text = menu;
            toolbarItem.DockStyle = BarDockStyle.Top;
            rootWorkItem.UIExtensionSites[
                UIExtensionSiteNames.MainToolbar].
                Add<Bar>(toolbarItem);
            rootWorkItem.UIExtensionSites.RegisterSite(
                toolbarName, toolbarItem);
        }
    }

    /// <summary>
    /// Adds the ribbon infrastructure if necessary. Pages and groups
    /// should be created prior to adding commands since more settings
    /// can be initialized. This is a last resort which will ensure
    /// that the pages and groups are created with default parameters
    /// if they do not already exist.
    /// </summary>
    /// <param name="page">The page on which this command will
    /// be displayed.</param>
    /// <param name="group">The group in which this command will
    /// be displayed.</param>
    private void AddRibbonInfrastructureIfNecessary(string page,
        string group)
    {
        string pageName = GetRibbonPageName(page);
        if (!rootWorkItem.UIExtensionSites.Contains(pageName))
        {
            RibbonPage ribbonPage = new RibbonPage(page);
            rootWorkItem.UIExtensionSites[
                UIExtensionSiteNames.MainMenu].
                Add<RibbonPage>(ribbonPage);
            rootWorkItem.UIExtensionSites.RegisterSite(
                pageName, ribbonPage);
        }
        string groupName = GetRibbonGroupName(page, group);
        if (!rootWorkItem.UIExtensionSites.Contains(groupName))
        {
            RibbonPageGroup ribbonGroup = new RibbonPageGroup(group);
            ribbonGroup.ShowCaptionButton = false;
            rootWorkItem.UIExtensionSites[pageName].
                Add<RibbonPageGroup>(ribbonGroup);
            rootWorkItem.UIExtensionSites.RegisterSite(groupName,
                ribbonGroup);
        }
    }

    private string GetRibbonGroupName(string page, string group)
    {
```

```
            return "Ribbon://Page." + page + "/Group." + group;
        }

        private string GetRibbonPageName(string page)
        {
            return "Ribbon://Page." + page;
        }

        private string GetMainMenuName(object menu)
        {
            return "MainMenu://" + menu;
        }

        private string GetToolbarName(string toolbar)
        {
            return "ToolBar://" + toolbar;
        }

        private RibbonItemStyles GetRibbonStyle(CommandRank commandRank)
        {
            switch (commandRank)
            {
                case CommandRank.Default:
                case CommandRank.Minor:
                    return RibbonItemStyles.SmallWithText;
                case CommandRank.Major:
                    return RibbonItemStyles.Large;
                case CommandRank.SubCommand:
                    return RibbonItemStyles.SmallWithoutText;
                default:
                    return RibbonItemStyles.Default;
            }
        }

        /// <summary>
        /// Creates a <see cref="DevExpress.XtraBars.BarButtonItem"/>
        /// and initializes the button properties that are common to
        /// both the ribbon and menu/toolbar.
        /// </summary>
        /// <param name="eventArgs">The
        /// <see cref="DevLynx.ShellApp.Infrastructure.Interface.
        /// UxExtensionEventArgs"/> instance containing the
        /// command data.</param>
        /// <returns>The <see cref="DevExpress.XtraBars.BarButtonItem"/>
        /// that was created and initialized.</returns>
        private BarButtonItem InitializeButton(UxExtensionEventArgs
            eventArgs)
        {
            BarButtonItem button = new BarButtonItem();
            button.Caption = eventArgs.Text;
            button.Glyph = eventArgs.SmallImage;
            button.LargeGlyph = eventArgs.LargeImage;
            return button;
        }
    }
}
```

To add a command a module would instantiate an IUxExtension, populate a UxExtensionEventArgs, and call the IUxExtension AddCommand method. An example:

```
public class ModuleController : WorkItemController
{
    public override void Run()
    {
        ExtendUx();
    }

    private void ExtendUx()
    {
        AddApplicationCloseCommand();
    }

    private void AddApplicationCloseCommand()
    {
        UxExtensionEventArgs eventArgs = new UxExtensionEventArgs();
        eventArgs.Text = Resources.ApplicationClose;
        eventArgs.CommandName =
            Constants.CommandNames.ApplicationClose;
        eventArgs.SmallImage = Resources.stop_16;
        eventArgs.LargeImage = Resources.stop_32;
        eventArgs.CommandRank = CommandRank.Major;
        eventArgs.Page = Resources.RibbonHomePage;
        eventArgs.Group = Resources.RibbonExitGroup;
        eventArgs.Menu = Resources.MenuFile;

        IUxExtension uxExtension = WorkItem.Services.
             Get<IUxExtension>(true);
        uxExtension.AddCommand(WorkItem, eventArgs);
    }

    [CommandHandler(Constants.CommandNames.ApplicationClose)]
    public void ApplicationClose(object sender, EventArgs e)
    {
        WorkItem.EventTopics[EventTopicNames.ApplicationClose].Fire(
            this, new EventArgs<string>(""), WorkItem,
            PublicationScope.Global);
    }
```

When the UxExtension.AddCommand method is called the service will determine which type of form is currently active and add the appropriate buttons to the user interface.

Another problem with a generic AppShell is how to handle application specific icons and captions. There are currently four things that we currently have to set for the AppShell: the application title; the document or project name that may be displayed on the caption bar; the application icon; and the image in the Ribbon's Application Menu. I have added an interface to both the ShellForm and the ShellRibbonForm to ensure that all of these items are taken care of. The IdevLynxCabMainApplicationForm interface:

```
namespace DevLynx.ShellApp.Infrastructure.Interface
{
    using System;
    using System.Drawing;
```

```
/// <summary>
/// Defines the methods required to show the application
/// specific information in the general purpose ShellApp.
/// </summary>
public interface IDevLynxCabMainApplicationForm
{
    /// <summary>
    /// Sets the application caption. Should be decorated with:
    /// <c>[EventSubscription(EventTopicNames.
    /// ApplicationCaptionUpdate, ThreadOption.UserInterface)]</c>
    /// </summary>
    /// <param name="sender">The sender.</param>
    /// <param name="e">The <see cref="DevLynx.ShellApp.
    /// Infrastructure.Interface.EventArgs&lt;T&gt;"/> instance
    /// containing the event data.</param>
    void SetApplicationCaption(Object sender, EventArgs<string> e);

    /// <summary>
    /// Sets the application document caption. Should be decorated
    /// with: <c>[EventSubscription(EventTopicNames.
    /// ApplicationDocumentCaptionUpdate,
    /// ThreadOption.UserInterface)]</c>
    /// </summary>
    /// <param name="sender">The sender.</param>
    /// <param name="e">The <see cref="DevLynx.ShellApp.Infrastructure.
    /// Interface.EventArgs&lt;T&gt;"/> instance containing the event
    /// data.</param>
    void SetApplicationDocumentCaption(Object sender,
        EventArgs<string> e);

    /// <summary>
    /// Sets the application ribbon icon. Should be decorated with:
    /// <c>[EventSubscription(EventTopicNames.
    /// ApplicationRibbonIconUpdate, ThreadOption.UserInterface)]</c>
    /// </summary>
    /// <param name="sender">The sender.</param>
    /// <param name="e">The <see cref="DevLynx.ShellApp.Infrastructure.
    /// Interface.EventArgs&lt;T&gt;"/> instance containing the event
    /// data.</param>
    void SetApplicationRibbonIcon(Object sender, EventArgs<Bitmap> e);

    /// <summary>
    /// Sets the application icon. Should be decorated with:
    /// <c>[EventSubscription(EventTopicNames.ApplicationIconUpdate,
    /// ThreadOption.UserInterface)]</c>
    /// </summary>
    /// <param name="sender">The sender.</param>
    /// <param name="e">The <see cref="DevLynx.ShellApp.Infrastructure.
    /// Interface.EventArgs&lt;T&gt;"/> instance containing the event
    /// data.</param>
    void SetApplicationIcon(Object sender, EventArgs<Icon> e);

    /// <summary>
    /// Closes the application.
    /// </summary>
    /// <param name="sender">The sender.</param>
    /// <param name="e">The <see cref="DevLynx.ShellApp.Infrastructure.
    /// Interface.EventArgs&lt;T&gt;"/> instance containing the event
```

```
        /// data.</param>
        void CloseApplication(Object sender, EventArgs<string> e);
    }
}
```

Both the ShellForm and the ShellRibbonForm implement this interface. From a different module these would be set using:

```
        protected virtual void SetApplicationInfo()
        {
            WorkItem.EventTopics[EventTopicNames.ApplicationCaptionUpdate].
                Fire(this, new EventArgs<string>(Resources.
                ApplicationCaption), WorkItem, PublicationScope.Global);
            WorkItem.EventTopics[EventTopicNames.
                ApplicationDocumentCaptionUpdate].Fire(this,
                new EventArgs<string>(
                Resources.ApplicationDocumentCaption),
                WorkItem, PublicationScope.Global);
            WorkItem.EventTopics[EventTopicNames.ApplicationIconUpdate].
                Fire(this, new EventArgs<Icon>(Resources.Gear_Blue),
                WorkItem, PublicationScope.Global);
            WorkItem.EventTopics[EventTopicNames.
                ApplicationRibbonIconUpdate].
                Fire(this, new EventArgs<Bitmap>(Resources.Collie),
                WorkItem, PublicationScope.Global);
        }
```

We now have an application which allows either a Ribbon or a menu/toolbar user interface. We have abstracted the creation of the UI extensions to ease this choice; and we have created CAB events to handle the application specific information such as the text in the application caption bar.

You will find the complete source code for this example at:
s3.amazonaws.com/DevLynx/DevLynx CAB 20071028.zip

dlx

Nov 4, 2007

More Generic Commands

In my last post I stated that Developer Express uses a common object for it's menus, toolbars and the Ribbon control, the BarButtonItem. This is not quite correct. The actual ancestor is the BarItem. The BarButtonItem covers the standard case (a button) but not all of the possibilities. We have to make a few changes to get more generic commands. We have abstracted the command structure with the UxExtensionEventArgs. The next step is to abstract which type of BarItem is created and inserted into the UI for the command.

Our goal is to replace references of BarButtonItem with BarItem and set up the infrastructure to allow various BarItem descendants to be instantiated. Not all of the BarItem descendants will work in the first iteration. We will make sure each descendent works as we develop them.

The first step is to change all references to BarButtonItem to BarItem with one exception: in the InitializeButton method of UxExtension leave the create alone for now. Shell app should still compile. I also changed all references to "button" to command or item.

The second step is to add a way to determine which descendant of BarItem to create. We will add a new property to the UxExtensionEventArgs class to store this information. In Delphi I would use:

```
TBarItems = class of BarItem;
```

For those not familiar with the Delphi construct "class of" refers to type metadata. We can now defer the decision of which BarItem descendent to a later time. This allows Delphi developers to easily implement an abstract factory pattern. Some (non-useful) Delphi code to do this is:

```
var
   ItemType : TBarItems;
   Item : BarItem;
begin
   ItemType := BarButtonItem;
   Item := ItemType.Create(); // we now have a BarButtonItem.
end;
```

In C# we use the generic Type class to store the item type and do a bit of runtime type checking:

```
private Type itemType;
/// <summary>
/// Gets or sets the type of the bar item for this command.
/// </summary>
/// <value>The type of the bar item for this command.</value>
public Type ItemType
{
    get { return itemType; }
    set
    {
        Guard.TypeIsAssignableFromType(value, typeof(BarItem),
```

```
                "BarItem");
            itemType = value;
        }
    }
```

Here I use the "Guard" class to ensure that the type passed in is acceptable during runtime. The Guard class is part of the CAB. Any class that is not assignable to a BarItem will cause an exception. Acalling method will be required to use typeof(BarButtonItem) to set the ItemType. There must be an elegant way to use Generics here but I'm not exactly sure how to accomplish that. If and when I figure it out I'll make modifications.

Now we have to instantiate this BarItem descendant within the UxExtension class. In the InitializeCommand method replace: `BarButtonItem button = new BarButtonItem();` with:

```
BarItem barItem = (BarItem)Activator.CreateInstance(
   eventArgs.ItemType, true);
```

Using the Activator.CreateInstance allows us to properly defer the decision of which BarItem descendant to employ until we are ready to create it.

Finally, we add a default to the UxExtensionEventArgs. I imagine the most oft used class will be the BarButtonItem. So we add the following line to the UxExtensionEventArgs constructor:

```
ItemType = typeof(BarButtonItem);
```

We can now use any of the BarItem descendants in our commands. Our next task is to create UIElementAdaptor's for the Ribbon ApplicationMenu and the RibbonPageHeader. The RibbonPageHeader is new with the Developer Express 7.3 release and allows adding of BarItems on the empty right side of the Ribbon Page tabs.

dlx

Nov 6, 2007

ApplicationMenu and RibbonPageHeader UIElementAdaptors

This weekend I did a little work on adding a UIElementAdaptor for the Ribbon ApplicationMenu. I also worked on a UIElementAdaptor for the RibbonPageHeader which is new to the Developer Express v2007 vol 3 beta. The RibbonPageHeader allows the addition of buttons and other BarItems to the right side of the Ribbon tabs area.

The code for the ApplicationMenuUIAdaptor:

```
using System;
using DevExpress.XtraBars;
using Microsoft.Practices.CompositeUI.UIElements;
using Microsoft.Practices.CompositeUI.Utility;
using DevExpress.XtraBars.Ribbon;

namespace CABDevExpress.UIElements
{
    /// <summary>
    /// An adapter that wraps an <see cref="ApplicationMenu"/>
    /// for use as an <see cref="IUIElementAdapter"/>.
    /// </summary>
    public class ApplicationMenuUIAdapter : UIElementAdapter<BarItem>
    {
        private ApplicationMenu applicationMenu;

        /// <summary>
        /// Initializes a new instance of the
        /// <see cref="ApplicationMenuUIAdapter"/> class.
        /// </summary>
        /// <param name="applicationMenu"></param>
        public ApplicationMenuUIAdapter(ApplicationMenu applicationMenu)
        {
            Guard.ArgumentNotNull(applicationMenu, "applicationMenu");
            this.applicationMenu = applicationMenu;
        }

        /// <summary>
        /// See <see cref="UIElementAdapter{TUIElement}.Add(TUIElement)"/>
        /// for more information.
        /// </summary>
        protected override BarItem Add(BarItem uiElement)
        {
            Guard.ArgumentNotNull(uiElement, "uiElement");
            if (applicationMenu == null)
                throw new InvalidOperationException();

            applicationMenu.AddItem(uiElement);
            return uiElement;
        }

        /// <summary>
        /// See <see cref="UIElementAdapter{TUIElement}.
        /// Remove(TUIElement)"/> for more information.
        /// </summary>
        protected override void Remove(BarItem uiElement)
```

```
        {
            Guard.ArgumentNotNull(uiElement, "uiElement");
            if (applicationMenu == null)
                throw new InvalidOperationException();
            if (applicationMenu.Ribbon == null)
                throw new InvalidOperationException();

            applicationMenu.Ribbon.Items.Remove(uiElement);
        }
    }
}
```

The code for the RibbonPageHeaderUIAdaptor:

```
using System;
using DevExpress.XtraBars.Ribbon;
using DevExpress.XtraBars;
using Microsoft.Practices.CompositeUI.UIElements;
using Microsoft.Practices.CompositeUI.Utility;

namespace CABDevExpress.UIElements
{
    /// <summary>
    /// An adapter that wraps a
    /// <see cref="RibbonPageHeaderItemLinkCollection"/>
    /// for use as an <see cref="IUIElementAdapter"/>.
    /// </summary>
    public class RibbonPageHeaderUIAdapter : UIElementAdapter<BarItem>
    {
        private RibbonPageHeaderItemLinkCollection ribbonPageHeader;

        /// <summary>
        /// Initializes a new instance of the
        /// <see cref="RibbonPageHeaderUIAdapter"/> class.
        /// </summary>
        /// <param name="ribbonPageHeader"></param>
        public RibbonPageHeaderUIAdapter(
            RibbonPageHeaderItemLinkCollection ribbonPageHeader)
        {
            Guard.ArgumentNotNull(ribbonPageHeader, "ribbonPageHeader");
            this.ribbonPageHeader = ribbonPageHeader;
        }

        /// <summary>
        /// See <see cref="UIElementAdapter{TUIElement}.Add(TUIElement)"/>
        /// for more information.
        /// </summary>
        protected override BarItem Add(BarItem uiElement)
        {
            Guard.ArgumentNotNull(uiElement, "uiElement");
            if (ribbonPageHeader == null)
                throw new InvalidOperationException();

            ribbonPageHeader.Add(uiElement);
            return uiElement;
        }

        /// <summary>
        /// See <see cref="UIElementAdapter{TUIElement}.
```

```
        /// Remove(TUIElement)"/> for more information.
        /// </summary>
        protected override void Remove(BarItem uiElement)
        {
            {
                Guard.ArgumentNotNull(uiElement, "uiElement");
                if (ribbonPageHeader == null)
                    throw new InvalidOperationException();

                ribbonPageHeader.Remove(uiElement);
            }
        }
    }
}
```

The code for initializing the controls in the application shell form:

```
    internal ApplicationMenu ApplicationMenu
    {
        get { return applicationMenu; }
    }

    internal RibbonPageHeaderItemLinkCollection RibbonPageHeader
    {
        get { return Ribbon.PageHeaderItemLinks; }
    }
```

Registering these UIElementAdaptors is done with:

```
    RootWorkItem.UIExtensionSites.RegisterSite(
        UIExtensionSiteNames.ApplicationMenu, new
        ApplicationMenuUIAdapter(this.Shell.ApplicationMenu));
    RootWorkItem.UIExtensionSites.RegisterSite(
        UIExtensionSiteNames.RibbonPageHeader, new
        RibbonPageHeaderUIAdapter(this.Shell.RibbonPageHeader));
```

There may be more things that could be done with the UIElementAdaptors but this is a good start.

dlx

Nov 17, 2007

Attempting to Abstract the UI Even More

The past couple of weeks I have been thinking about abstracting the UI even more. If I can successfully abstract for both a Ribbon and standard menu paradigms, why can't I extend that to handle the standard Microsoft menu system and other control sets? In this scenario each of the possible shell applications and shell forms would be in separate projects. If the Developer Express controls were not available then it is a simple matter to remove the projects which depend on these controls. It was an interesting thought exercise, but in the end the application would be restricted to the lowest common denominator for the UI. The goal of my project is to see if the CAB could be used to create a rich user interface. So I abandoned the majority of this thought exercise.

I did implement one section to further abstract the UI. The ShellMain and UxExtension service now use Reflection to determine which interface paradigms were available and to further abstract the invocation of the selected paradigm. In previous versions of the ShellApp I would read the boolean app setting "IsRibbonForm". IsRibbonForm would determine if the ShellRibbonApplication or the ShellApplication was launched. But as a boolean it would never handle more than these two options.

In short, the solution is to decorate descendants of SmartClientApplication with an attribute (ShellApplicationAttribute). We also separate the Ribbon and menu paradigms in the UxExtension service and decorate the extracted classes with an attribute (UxExtensionAttribute). The attribute code is:

```
[AttributeUsage(AttributeTargets.Class)]
public class ShellApplicationAttribute : Attribute
{
    public ShellApplicationAttribute(string name, string description)
    {
        this.name = name;
        this.description = description;
    }

    private string name;
    public string Name
    {
        get { return name; }
    }

    private string description;
    public string Description
    {
        get { return description; }
    }
}

[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
public class UxExtensionAttribute : Attribute
{
```

```
        public UxExtensionAttribute(string name)
        {
            this.name = name;
        }

        private string name;
        public string Name
        {
            get { return name; }
        }
    }
```

The name property of each of the attributes is used to determine which UI paradigm to use. This name property is read from the application configuration file. The shell assembly will then be searched for a class decorated with the ShellApplicationAttribute with the name from the configuration file. The ShellMain class will then invoke the run method of the class. Here is the new code for ShellMain:

```
namespace DevLynx.ShellApp.Infrastructure.Shell
{
    using System;
    using System.Configuration;
    using System.Collections.Specialized;
    using System.Reflection;
    using DevLynx.ShellApp.Infrastructure.Interface;
    using System.Collections.Generic;
    using System.Text;

    /// <summary>
    /// Main application entry point class.
    /// </summary>
    class ShellMain
    {
        /// <summary>
        /// Application entry point.
        /// </summary>
        [STAThread]
        static void Main()
        {
            DevExpress.UserSkins.BonusSkins.Register();
            DevExpress.UserSkins.OfficeSkins.Register();
            DevExpress.Skins.SkinManager.Default.
                  RegisterAssembly(typeof(DevExpress.UserSkins.
                  Office2007Bonus).Assembly);

            NameValueCollection appSettings =
                  ConfigurationManager.AppSettings;
            SetEnableFormSkins(appSettings);

#if (DEBUG)
            string modeMethod = "RunInDebugMode";
#else // (RELEASE)
            string modeMethod = "RunInReleaseMode";
#endif // if (DEBUG)

            MethodInfo runMethod = GetApplicationType(appSettings).
                  GetMethod(modeMethod,BindingFlags.NonPublic
```

```
            | BindingFlags.Public | BindingFlags.Static);
        if (runMethod == null)
        {
            // TODO: Log this exception
            throw new InvalidOperationException(
              "Could not get the " + modeMethod
              + " method to launch the application.");
        }
        runMethod.Invoke(null, null);
    }

    private static Type GetApplicationType(
        NameValueCollection appSettings)
    {
        // We have decorated the ShellApplication classes
        // with the ShellApplicationAttribute. So now find the
        // ShellApplicationAttribute with the corresponding
        // UxExtensionType name. We will then call the run method
        // of that class. This  gives us some flexibility to
        // add more ShellApplications with different UI components
        // when we need them. If we want to add a ShellApplication
        // from a different assembly than
        // DevLynx.ShellApp.Infrastructure.Shell we will have to
        // interogate other assemblies for the
        // ShallApplicationAttribute.

        // Store all of the possible values for UxExtension so that we
        // can display the valid values in an error message.
        List<string> acceptableUxExtensionTypes = new List<string>();

        string uxExtensionType = appSettings.Get("UxExtensionType");
        Type applicationType = null;

        // Now get all of the classes that are in this assembly.
        // We will iterate through each class and get any
        // ShellApplicationAttribute that is decorating the class.
        // When we find the attribute with the UxExtensionType from
        // the configuration file we store it for return to the
        // calling method.
        System.Type[] existingClasses=Assembly.GetExecutingAssembly().
              GetTypes();
        foreach (Type type in existingClasses)
        {
            object[] classAttributes = type.GetCustomAttributes(
                    typeof(ShellApplicationAttribute), false);
            if (classAttributes != null && classAttributes.Length > 0)
            {
                string name = (classAttributes[0]
                    as ShellApplicationAttribute).Name;
                // store in case we have to throw the exception below
                acceptableUxExtensionTypes.Add(name);
                if (String.Compare(name, uxExtensionType, true) == 0)
                {
                    applicationType = type;
                }
            }
        }
```

```
            // if we didn't find a class that has been decorated with
            // the appropriate attribute and name then we are done.
            // Throw an exception and get out.
            if (applicationType == null)
            {
                //TODO: log the exception
                StringBuilder stringBuilder = new StringBuilder();
                stringBuilder.Append("There is an invalid UxExtensionType
                   value in your configuration file. Current value: ");
                stringBuilder.Append(uxExtensionType);
                stringBuilder.Append(". Valid values are: ");
                foreach (string value in acceptableUxExtensionTypes)
                {
                    stringBuilder.Append(value);
                    stringBuilder.Append(" ");
                }
                throw new
                    InvalidOperationException(stringBuilder.ToString());
            }
            return applicationType;
        }

        private static void SetEnableFormSkins(
            NameValueCollection appSettings)
        {
            bool enableFormSkins = false;
            try
            {
                string formSkins = appSettings.Get("EnableFormSkins");
                enableFormSkins = Convert.ToBoolean(formSkins);
            }
            catch (FormatException ex)
            {
                // TODO: log this format error will use ex at that time
                // If this is a format error then the EnableFormSkins
                // app setting is probably not either "true" or "false".
                // Eat the exception and keep the default set above.
            }
            if (enableFormSkins)
                DevExpress.Skins.SkinManager.EnableFormSkins();
        }
    }
}
```

Similar code is used to determine which UxExtension class we want to instantiate.

dlx

Nov 19, 2007

Status Update and Reality Check

I started this investigation into CAB about two months ago now and it's time to review the process and determine if CAB is still meeting the objectives.

First, as so many people have already said, there is a steep learning curve to CAB and the SCSF. I have been working with it for two months and I have only scratched the surface. Granted, I have only been developing in my spare time; but I still do not have a complete grasp of how all of the pieces fit together yet. This work will be time well spent if we do use the CAB for the big rewrite at the day job. It's too complex for my group to figure out without one person already having knowledge of the system.

There is no built-in ability to control order when adding items to the UIExtensionSites. In the current iteration of my test program the application exit command is always the first command in the menu; very non-standard. David Platt, in his book Programming Microsoft Composite UI Application Block and Smart Client Software Factory, describes one way to handle this problem: 1) Create an event to which all modules subscribe. 2) Some looping mechanism fires that event with the name of a module as a parameter. 3) When a module sees its name it adds its UI elements.

Can we say kludge? I think that the CAB should have accounted for this functionality. Unless I can think of a better way, I'll be using a variation of this method - but I don't like it.

The CAB creates loosely coupled modules that are composed into a rich application. Through guidance packages, the SCSF extends the functionality of the CAB. It makes perfect sense that a development team will create modules of an application in separate solutions. So why is it that you cannot create a Business Module in a separate solution from the shell? That's right, you get an exception when running the Add Business Module recipe. The wording is: "An error happened while calling the value provider or evaluating the default value of argument ShellProject." There is no reason for this restriction. I have a temporary solution with a shell where I create Business Modules. I then copy the code verbatim into my actual project; the code works without modification. The option to add a View is not even available in a solution without a shell. What the...? This is a gross oversight. Eventually I'll create Visual Studio templates to add modules and views - I shouldn't have to.

And yet, there is power here. Decoupling to this degree is the right way to develop. Will it work for me and my crew in the big rewrite? I don't know yet. But it's worth further investigation; and I'm having a blast learning a new technology. This is why I love my job!

dlx

Dec 2, 2007

Status Update

It's been two weeks since my last entry. I have been busy with the day job as well as other personal matters. So, I have not had a great deal of time to contribute to the project. Some of the issues that I have worked on:

- Removing the Source/Infrastructure directories in the ShellApp. Since the ShellApp is going to be shared and no other modules placed into the solution it seemed wrong to have the additional directory levels. So I flattened out the directory structure.
- In the previous released source of the ShellApp I had placed our standard copyright notice from the day job. This has been replaced with the MIT License so that others can legally use my additions.
- Added structure to centralize some of the common command images that are invariably in most projects. This way when someone demands something silly like "We must change the 'cut' command image to a bloody butcher's knife?" it can be done in all projects at once.
- Added command overlay images. At work I always seem to be the one who creates command images. I know, we should get a graphic artist to do this - and actually we have access to one now! I have spent untold hours creating various images for our commands (they were adequate but I am not an artist). One of the things that takes so much time are what I call overlays. An overlay is a small command image modifier such as a plus sign for 'add'. For instance a 'Project' command will have modifiers for add, delete, edit, import, export, make it spin around on it's head three times, etc. I always do this with a base image and the modifier is placed in the lower right hand corner. When someone asked for an overlay replacement I would have to go in and change many command images. Now overlays can be placed on the image automatically, are easy to change, and are consistent across commands and projects.
- Created automated builds using FinalBuilder.
- I have set up FogBugz to use as an issue tracking system and have started to track issues.

In other words, I have not been completely idle. My next step is also going to be more structure. I will be adding unit tests and doing code coverage. So it probably won't be until after the holidays that I have more features implemented.

dlx

Dec 8, 2007

Source Code Release

I have consolidated all changes to date for another source code release. Since my work on the CAB and Developer Express components is a work in progress there will be breaking changes in this code. Most of the code releases that I will publish in the next months will have breaking code changes. I apologize in advance for this inconvenience.

This release includes all changes since the last source code release, the items mentioned in my Dec 2, 2007 entry as well as an additional UIElementAdapter for the Ribbon Quick Access Toolbar. I also changed the name of the ApplicationMenuUIAdaptor to RibbonApplicationMenuUIAdaptor so that all of the Ribbon centric classes are prefixed with 'Ribbon'.

The RibbonQuickAccessToolbarUIAdapter code follows:

```
using System;
using Microsoft.Practices.CompositeUI.UIElements;
using DevExpress.XtraBars;
using DevExpress.XtraBars.Ribbon;
using Microsoft.Practices.CompositeUI.Utility;

namespace CABDevExpress.UIElements
{
    public class RibbonQuickAccessToolbarUIAdapter : UIElementAdapter<BarItem>
    {
        private RibbonQuickAccessToolbar ribbonQuickAccessToolbar;

        public RibbonQuickAccessToolbarUIAdapter(RibbonQuickAccessToolbar
              ribbonQuickAccessToolbar)
        {
            Guard.ArgumentNotNull(ribbonQuickAccessToolbar,
                    "ribbonQuickAccessToolbar");
            this.ribbonQuickAccessToolbar = ribbonQuickAccessToolbar;
        }

        protected override BarItem Add(BarItem uiElement)
        {
            Guard.ArgumentNotNull(uiElement, "uiElement");
            if (ribbonQuickAccessToolbar == null)
                throw new InvalidOperationException();

            ribbonQuickAccessToolbar.ItemLinks.Add(uiElement);
            return uiElement;
        }

        protected override void Remove(BarItem uiElement)
        {
            Guard.ArgumentNotNull(uiElement, "uiElement");
            if (ribbonQuickAccessToolbar == null)
                throw new InvalidOperationException();

            ribbonQuickAccessToolbar.ItemLinks.Remove(uiElement);
        }
    }
}
```

As I mentioned in the Dec 2, 2007 entry, I have added support for both common command images as well as command overlays images. Images that I use have been purchased from glyFX and, therefore, I cannot release the actual images. I have blurred the images (using ImageMagick) so that I could include placeholders in the source distribution. You will have to replace these blurred images with your own if you choose to use the command images and overlays.

You will find the source code at:
http://s3.amazonaws.com/DevLynx/2007 12 08 DevLynx CAB Deployment.zip

dlx

Jan 8, 2008

NUnit, NMock2 and Sandcastle

It's been a month since my last update. The holiday season is over and I'm back to work on my CAB test application. These past few weeks I have been focusing on unit testing, documentation and automated builds. I have also added one service, the ConfigurationStore.

The ConfigurationStore allows the user to generically read and write type safe values to configuration files in a more structured way then the standard appSettings. It allows multiple stores with categories and values - basically a two organizational levels and a terminal nodes to store key-value pairs. Currently types supported are: Bool, DateTime, DateTimeUtc, Double, Enum, Integer, and String. I will add more as they become required. Since I am testing the CAB for multiple large scale projects the ConfigurationStore can manage information for the following situations:

- AppConfig: application specific values that the user should not change (we are assuming that the user does not have access to the application location so we don't store values that should be user modifiable in the application configuration file).
- SharedCommon: values which are shared between users and common to all products.
- UserCommon: values which are user specific and common to all products.
- SharedProduct: values which are shared between users and product specific.
- UserProduct: values which are user specific and product specific.
- UserCustomization: values which are user specific and holds user customizations (such as window placement, etc).

The ConfigurationStore class has 100% code coverage through NUnit tests. This is my first real attempt at unit testing (I know, I'm a late bloomer :-). I'm sure that all paths are not covered but I hope that will be resolved as I learn better testing techniques.

Overall unit testing is up to date. I have unit tests on much of my code but have excluded unit tests on classes that the SCSF created. The code that does not have unit tests are the Shell and those that must interact with the WorkItem.UIExtensionSites. The CAB code included a TestableWorkItem that recreated some of the WorkItem but it did not include the UIExtensionSites. I may be able to work something out at a later date but for now I'm going to continue.

I also experimented with mock objects for the first time. This was a bit frustrating since NMock2 does not have the best documentation; but I was pleased with the final result. All of the tests will be included in the next source release.

I have also started to smooth out the XML documentation of classes. It's by no means complete but it is usable. I'm using Microsofts SandCastle with Eric Woodruffs Sandcastle Help File Builder to produce a help file. I have also experimented with adding additional

content to the generated help files so that I can include a framework overview and task oriented entries into the help file. I'm going to say something that is perhaps not so popular when dealing with open source software. My job is to create software for sale. I don't have time to peruse the code of an open source tool to figure out how it works-I want documentation. If a development tool has insufficient documentation then it's not going to be used. Documentation is important to me and I'm going to supply it.

So where to next? I have enough structure in place to start the actual test application; the DevLynx Address Book. The next task is to set up a database and start using the framework. As framework issues arise I'll make modifications to ensure that it meets the goals.

dlx

Jan 20, 2008

Refactoring, Sandcastle, Data Access, and Source

This is the first release in which I have included the sample program, DevLynx Address Book (DAB). The DAB has all of one function, it will allow you to enter contact names. It uses the new DataAccess service which centralizes some of the common data access needs; more on that later. The DAB comes with source and a zip file which includes the running executable.

Here are some of the new features/refactorings: First I removed the "Infrastructure" level of the ShellApp. The SCSF added the infrastructure level as an organizational method anticipating that business modules would be in the same solution. Since that is not the case with the ShellApp and everything was in the infrastructure directory I felt it needed to be removed.

I found and added the Versioning Controlled Build application to help synchronize the version numbers. This tool has both a Visual Studio add-in and a command line tool. I added the command line tool to the FinalBuilder automated build and now I no longer have to think about versioning any builds.

As soon as I started the DAB sample application I immediately needed a way to persist the data. I wrote the DataAccess service which handles connection string storage, concurrent connections to multiple databases and in memory data sources. The DataAccess service is based on the DevExpress eXpress Persistent Objects (XPO) which allows connecting to a number of different databases without code changes (XPO is an Object Relational Mapping tool). I have tested it with the in memory data source, VistaDB3, Access and SQL Server.

One of the nice features of XPO is support for an in memory data source. It stores persistent data in a .NET DataSet but allows access via the same method as other databases. The DataAccess service can use the in memory data source and includes the ability to save and load the dataset from the disk. This was first developed for testing speed but will have use in general purpose development.

The final functionality was to create additional content for the DevLynx ShellApp help files. There is now an overview topic discussing the functionality of the ShellApp. I will add examples from the DAB in the future. The additional content is simply HTML files included by the Sandcastle Help File Builder. However, I wanted the additional content to use the same CSS files, scripts and images and look and feel of the XML comment generated source. After a bit of experimentation it seems to be working great. I encountered two problems. First Sandcastle is SLOW. It takes about nine minutes to generate the help files. So every time I needed to test my new content I would have to wait. I hope that Microsoft speeds up the processing a bit. I looked around for a free HTML editor that

would satisfy my simple needs. After a bit of searching I settled on Komposer. I worked with it for a few hours but finally got tired of it reformatting my HTML source into something unreadable. I set the option to keep the original formatting but it didn't matter. So I finally downloaded a trial copy of TopStyle by Nick Bradbury. It provides few frills but it works and doesn't try to "help me out".

To run the DAB all you need to do is to unzip the files and run the executable. It places nothing in the registry and does not register anything on your computer. It does, however, place configuration files and its database into the directories pointed to by `Environment.SpecialFolder.CommonApplicationData` and `Environment.SpecialFolder.ApplicationData`. To change the database used by the DAB you can modify the SharedProductSettings.config file in the CommonApplicationData directory. Just change the "Primary" stringVaue connection string link to point to one of the other connection strings in the ConnectionString category. You can look into the connection strings group to see the different databases that have been tested. If you use the SQL Server connection string you will need to insert your own server and instance names.

```
<category name="ConnectionStringLinks">
    <value name="Primary" stringValue="VistaDB3" />
</category>
```

You can change to the menu/toolbar paradigm by changing UserCommonSettings.config in the ApplicationData directory. Just change UxExtensionName stringValue from DxRibbon to DxMenu.

```
<category name="UxSettings">
    <value name="UxExtensionName" stringValue="DxRibbon" />
</category>
```

You will find the source code at: http://s3.amazonaws.com/DevLynx/2008 01 20 DevLynx CAB Source.zip and the DAB binary at: http://s3.amazonaws.com/DevLynx/2008 01 20 DevLynx CAB.zip

dlx

Jan 28, 2008

Protecting a decoupled ShellApp and the splash screen

One of the features that I added for the last release was a ProductInformation.config file. This configuration file stores information about the company, product and splash screen. This information is accessed via the ProductInformation service. From the help content:

> The DevLynx ShellApp is designed to be used with multiple applications without recompiling. However, the ShellApp need to know some information about the product including the company name, product name, paths where product specific information is to be stored, splash screen image, etc. This information must be decoupled from the ShellApp. The product information service addresses this need.
>
> The product information service reads the appropriate information from a configuration file. This file is named "ProductInformation.config" and must reside in the same directory with the application executable. The configuration file is a standard app config file with an appSettings section.
>
> The ProductInformation.config file is designed to never be changed by the user or the application. So only place information in this file that is constant for the lifetime of this release.

Having a text file with your company name and splash screen may be an enticement for your users. They may want to change the company name or the splash screen image. So how do we go about protecting this information from the "vandalism" of users?

The ProductInformation service has the capability to store an MD5 hash for a file and then check that value to ensure that the file has not changed. This can be done for the splash screen image. But of course the ProductInformation.config file must be protected as well. Here we must store the pre-computed MD5 hash of the ProductInformation.config file somewhere else and check it against the file. The ValidateProductInformation method takes an MD5 hash string as a parameter. The stored string can be located anywhere. In the DevLynx Address Book I have it stored directly in the main Address Book assembly which can be strongly named as well. Of course if you don't care if someone makes changes to these files then you don't have to use these capabilities.

I also added a splash screen to the shell since I felt that the pause between launching the application and the shell appearing was too great. The splash screen is a modification of the asynchronous splash screen at: http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=27DF8F87-37A7-4751-8198-B9A3526BBCDD This is a general purpose splash screen and all you have to provide an image to display and a few properties and the rest is done automatically. If

you don't provide an image it will still run but as a thin window with the status text displayed. The splash screen options are stored in the ProductInformation.config file and are listed below (again straight from the help content):

- SplashImageFileName - Contains the file name of the image that will show on the splash screen. If this name is blank then ShellApp will look for DevLynx.ShellApp.Shell.exe.png for the splash screen image.
- SplashImageFileMD5 - If this value is present the ShellApp will check to ensure that someone has not altered the splash screen image file. It must contain the MD5 hash of the image file.
- SplashStatusLocation - The X, Y coordinates of the splash screen status text. The default location will be 10 pixels from the left and 10 pixels from the bottom of the image.
- SplashTextColor - The color of the status text on the splash screen. The default is black. Colors must be one of the named colors from the System.Drawing.Color class.
- SplashTransparencyKey - The portions of the image which are this color will be completely transparent. Note that partial transparency is not supported here. Hard edges are required to make this look good. Colors must be one of the named colors from the System.Drawing.Color class.
- SplashOpacity - Setting the SplashOpacity to something less than 1.0 will make the splash screen transparent. This number needs to be between 0 and 1.0.
- SplashIsHaloText - Since we are placing the status text directly on an image the text may blend into the image and be unreadable. By setting the SplashIsHaloText to "true" the status text will surrounded by a halo effect in the SplashTextHaloColor. Note that if the halo text goes into a transparent section of the splash screen it will not look very good.
- SplashTextHaloColor - The color of the halo surrounding the status text on the splash screen. The default is WhiteSmoke. Colors must be one of the named colors from the System.Drawing.Color class.

Using these options you can develop nearly any splash screen that is desired. I'll admit I got a little carried away with the splash screen options. I always wanted to see an algorithm for halo text (http://www.bobpowell.net/halo.htm) and once I found one I had to implement it.

I also continued to develop the help content. While it is far from complete I am pleased with what I have to this point. The next step is to write the help content for all of the features that I have developed so far - time to play catch up. As with many developers, I would like to continue adding features; but if I don't catch up with the help content now it will be too daunting a task.

dlx

Jan 28, 2008

Sandcastle January 2008 Release

I downloaded the Sandcastle January release earlier and just checked the speed. Compiling the DevLynx ShellApp help content took 9 minutes with their previous release; it now takes 1:48. Very nice work! Wow.

dlx

Feb 17, 2008

Documentation and Source Code Release

Well, I was right; documentation is a daunting task. Even the little bit that I have produced took much longer than expected - I have a whole new respect for technical writers. I don't have everything documented but I do have a good start and will hopefully not fall behind again.

There have been very few changes to the code base since adding the splash screen.

- ConfigurationStore.CreateStore now writes cleaner XML into the configuration section.
- I have added a DefaultUxParadigm to the ProductInformation.config file. This value allows you to select which paradigm comes up when the application is first initialized.

The latest code and PDF of the blog can be found on my web site http://www.tetzel.com.

Next steps:

- Keeping the MD5 hash codes in both the ProductInformation.config file and the ValidateProductInformation is too prone to error. I will be creating automated build tools which will allow these items to be set during the build process. There will be a series of console applications that can be run from batch files or integrated into build tools such as FinalBuilder.
- The automated build tools will rely on command line arguments. I looked around the internet for a class that I could use. However, they were either too simple or more complicated than I wanted. So I will write a class to manage command line arguments.
- I will also work on a feature to allow some rudimentary ordering of UxExtension elements. Right now my Exit menu item always appears at the top of the menu which would be very confusing for users expecting to see it at the bottom.

dlx

Feb 24, 2008

How do I compile the DevLynx ShellApp?

Based on a comment by manayat regarding my Feb 17th post, I thought it would be a good time to describe how to use the DevLynx ShellApp. So this weekend I created a virtual machine and loaded Visual Studio 2005 and the Developer Express components. I then documented how to get both the ShellApp and Address Book to compile. There were some modifications to make it easier for the first time user. There are way too many steps but I will hopefully automate some of them as the product gets more mature.

The documentation is in the help file found in the Source\ShellApp\Support\Help directory. Go to the task-based help and view the articles there.

This code (along with the help file) and a PDF of the blog can be found on my web site at http://www.tetzel.com.

dlx

March 9, 2008

Progress Update

I had to leave town for a week on personal business so I didn't get as much done as I would have liked. I have completed the class structure to manage command line arguments. I now need to create the automated build tools for managing the MD5 hash information. After that I will update the documentation to include the command line arguments classes. As soon as those tasks are complete I'll do another source code release.

As I have said, we may not use the DevLynx ShellApp at work. However, we will almost certainly be using the ConfigurationStore and the command line arguments classes. So I have separated these general purpose classes into a DevLynx.Utilities solution. That way they can be used without the ShellApp.

I hope to have a source code release next weekend.

dlx

April 17, 2008

Source Code Release

Well, from March 9, next weekend extended into nearly 6 weeks. Too many irons in the fire extended the planned release date a bit (OK, quite a bit).

This release includes a couple of new features. I have separated the non-CAB specific functionality. The DevLynx.Utilities solution includes the ConfigurationStore, the DataAccess class, the new CommandLineArgument classes, and various helper classes. This new solution allows developers to access the auxiliary functionality without having to include the ShellApp and CAB. I did not, however, separate the help content for these non-CAB specific items.

The basics for command line argument support is included in this release. From the ShellApp documentation:

Command line arguments are a useful tool for temporarily modifying application options; however, configuration files should be used for any long term option changes. DevLynx.Utilities contains a simple method for handling command line arguments relying on .NET attributes and reflection.

The syntax for DevLynx.Utilities compliant command line arguments is as follows:

```
[/|-|--]key[=|:]value | key
```

key is an identifier of any length greater than 1 containing only a-z, A-Z and 0-9. value may be either an identifier with the same restrictions as the key or it may be a string surrounded by double quotes (“) or a string surrounded by single quotes (‘). The optional prefix can be one of slash, dash or double dash. No space is allowed between the prefix and the key. If there is a value it must be separated from the key with a delimiter of an equal or colon character. No spaces are allowed between the key and the delimiter or the delimiter and the value. If the key has no corresponding value it is treated as a boolean value (it either exists or does not exist within the command line arguments).

The DevLynx.Utilities command line arguments are not positional; they cannot be programmatically accessed via an index. Instead they can be used in one of two ways:

- They may be programmatically accessed via the key identifier. In this case the value will be returned as a string.
- The preferred method is to create a class with properties decorated with the CommandLineArgumentAttribute attribute. The CommandLineArguments class will then fill the class properties with the corresponding values. In this case the value will be type safe and validated for the requested type.

A simple validation method may also be included in the data class of the DevLynx.Utilities command line arguments. This method will be called as soon as the instance of the data class is loaded.

Additional Restrictions:

- Duplicate key entries are ignored. The first argument with the key will set the value.
- Strings may not contain the character with which they are quoted.
- key entries are case insensitive.
- Boolean arguments without a value element will not change their corresponding properties to false if the argument does not exist on the command line; the value will remain as set in the class instance.

Currently, the command line arguments support only boolean and string values. Future enhancements will handle additional types.

There may be some readers who are interested in the ShellApp but don't want to invest the time to examine the code at this early stage. I have uploaded a web version of the documentation so that you can see the current project state without downloading.

The current source code release, the web based documentation, and a PDF of the blog can be found on my web site at tetzel.com.

dlx