Hosting Windows Forms Designers - 14 June 2003

Back to Articles

Revisions

Introduction

I decided to write this article not because there is a strong demand for this information, but because there is literally no existing information out there on the topic. The documentation is scarce if any, and aside from a few tidbits thrown out by Microsoft it is a daunting task. It requires you to already be very familiar with the design-time architecture, and have a strong grasp of all the interfaces you commonly use.

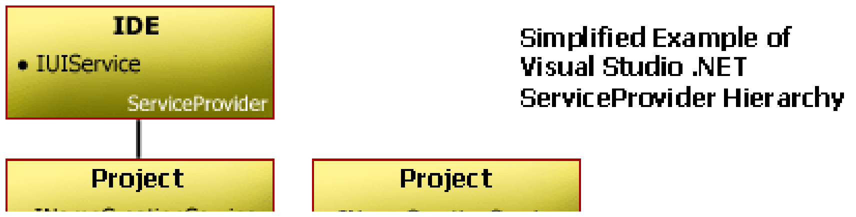
All the interfaces you rely on when developing designers and extending the design-time behaviour of your controls, it is now your responsibility to implement. Not only is this a non-trivial task, but it is one of those times where you have to implement a great deal before you can see any progress at all. That said, once you've written the code you'll never have to write it again (and since I'm going to be writing it for this article, you'll never have to write it in the first place).

Why would you want to host the Windows Forms Designers? Well, there could be lots of reasons. When I was faced with the problem it was because I was writing an IDE for .NET languages and I wanted a visual interface to configure GUIs, just like VS.NET has. When you consider the flexibility of the designer architecture, you realise you have a framework for designing any 2-dimensional hierarchical system. All the code for moving, resizing, creating, deleting and configuring items is already there.

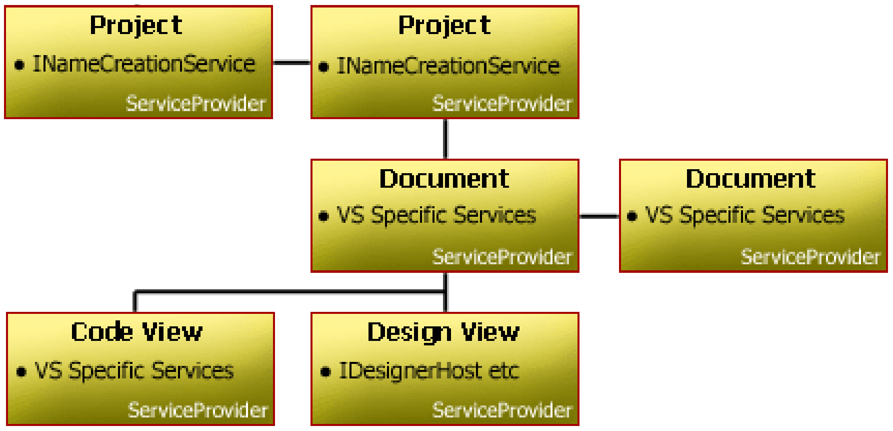
The one you see in VS.NET is running from the framework, and is the same one we'll be writing the code to enable. All the designer stuff is already present in the framework, you just need to write the code to bind it all together. This was best left out of the framework because it is strongly tied to the host environment.

The Service Hierarchy

The most fundamental thing to be familiar with on this topic is the service hierarchy. You have already used it, whenever you make a call to


IDE
• IUIService
ServiceProvider
Project
Project
Simplified Example of Visual Studio .NET ServiceProvider Hierarchy

GetService to obtain an instance of ISelectionService or IComponentChangeService. A service hierarchy is made up of several ServiceContainers. A ServiceContainer is used to store an instance of a class that provides a service against its type. So you request a instance by passing it a type, and it comes back with what you need.



When a ServiceContainer is created it is usually bound to a parent. This means that when a service is requested and it isn't in one container, it asks its parent, and then that asks its parent, and so on. ServiceContainers have no concept of children, only of parents. Using the example of the Visual Studio IDE, services are instantiated at several levels. At the document view level you have services such as ISelectionService and IComponentChangedService. These are obviously only relevant to one document (and only the design view, not the code view).

Look at the ITypeResolutionService interface though. Chances are you'll never use it, it's how the designers discover and instantiate types as they deserialize a document. Although the view-level servicecontainer will get the request for it, it will be the servicecontainer at the project level that will handle the request. Visual Studio is made up of a hierarchy of servicecontainers, from the toplevel ones that are used internally to manage menus and colours, to the document and view-level ones for editing documents. The image on the right shows a simplified look at part of this gierarchy.

It's very important when hosting designers to get the servicecontainer hierarchy right - it can make things more logical and therefore more understandable. Always remember that the way everything at design time works together is by getting and using services from the host environment.

The Design Container

No matter what you're designing, be it Windows Forms or Web Forms, everything in design mode is said to be on the *Design Surface*. What this means is that it's in the design container. Forms and controls are hierarchical in nature, but everything is also in the design container, which is a one-level collection. This container has the code to discover and instantiate designers for objects, and also dispose of them when they're done.

As you probably already know, all designers implement the IDesigner interface. However, there is always one special designer in a document which has to implement IRootDesigner. This is the one that acts as a parent to all the rest, and is responsible for providing the interface you use to view the object you're designing. This is always the first object to be added to the design container. The framework provides three root designers - for Forms, UserControls and Components. You've probably seen all these if you've used the various designable types in Visual Studio. The framework doesn't contain any Web Forms root designers.

The root designer is a great idea, it means you don't have to write any special hosting code no matter what you're designing. Once your hosting framework is in place you can design anything that offers one.

The Designer Host

This is where you start when hosting designers (obviously). It all begins with the IDesignerHost interface, which you might well have already used from designers you've written. It exposes the design container, the root component, and methods for creating components and getting their designers. It also provides the support for designer transactions and acts as a service container.

Although we'll be implementing quite a few interfaces apart from this one, the most important two are ISelectionService and IComponentChangeService, because every designer uses them. Thankfully most of the .NET controls behave gracefully when they fail to find a service implemented, if they didn't we would have a much harder time getting started.

The core designer hosting architecture is split in to several interfaces, but since they are tightly bound together we will be implementing the most important ones all in one class:

- *IDesignerHost* - The core interface.
- *IContainer* - The container that holds all components on the design surface.
- *IComponentChangeService* - Used to broadcast events when components change.
- *IExtenderProvider* - Used to give components a configurable name.

Design Sites

In order to be on the design surface, a class has to implement IComponent. This means it has a Site property, which is how its name is kept track of at design time. For every object that is placed on the design surface an object implementing ISite needs to be assigned to it. It's through this that the components are able to request services, establish that they are in design mode and get their name.

We will create a class, DesignSite, that implements ISite and provides this necessary information. Anyway, enough with the theory, let's get down to business.

Starting Off

First we'll create our class, DesignerHost, and make sure we have a reference to System.Design.dll. A lot of the interfaces we'll be using are in there. We will accept an instance of something implementing IServiceContainer as the parameter to the constructor, and since we have to provide an

IServiceContainer implementation ourselves we'll simply wrap that one. Since that's the easiest part we'll do it first.

```
public object GetService(System.Type serviceType)
{
        return parent.GetService(serviceType);
}

public void AddService(System.Type serviceType,
System.ComponentModel.Design.ServiceCreatorCallback callback, bool promote)
{
        parent.AddService(serviceType, callback, promote);
}

public void AddService(System.Type serviceType,
System.ComponentModel.Design.ServiceCreatorCallback callback)
{
        parent.AddService(serviceType, callback);
}

public void AddService(System.Type serviceType, object serviceInstance, bool promote)
{
        parent.AddService(serviceType, serviceInstance, promote);
}

public void AddService(System.Type serviceType, object serviceInstance)
{
        parent.AddService(serviceType, serviceInstance);
}

public void RemoveService(System.Type serviceType, bool promote)
{
        parent.RemoveService(serviceType, promote);
}

public void RemoveService(System.Type serviceType)
{
        parent.RemoveService(serviceType);
}
```

At the same time as implementing IDesignerHost we will implement IContainer. This is the design container I spoke of before. All the code we're writing really revolves around the code that gets and instantiates designers for the objects added to the host.

IContainer.Add

This method has two overloads, one where a name is passed and one without. We'll just defer the latter to the former, passing null and dealing with it correctly. This is one of the methods with the real meat in.

The first thing we do after checking the passed component isn't null, is to check the component isn't already sited somewhere. If it is, we remove it from its current container. Then we generate a name for the component if we haven't been passed one, using INameCreationService. We'll come on to implementing this interface later. Next we make sure there isn't already a component with the passed name in the container.

The next stage is to give the new component a Site, which we do with our ISite implementation. I'll come on to that later. Giving it a site as early as possible is important, because that's how the component is able to request services from its environment. Then we make the important call to TypeDescriptor.CreateDesigner, which actually finds and created the designer associated with this component. If this is the first component being added we look for an IRootDesigner and set the rootComponent field.

Next comes a check to make sure we got a designer, and if so we initialize it. Remember the Initialize method of a designer that you override when making your own? This is where we actually call that method. After this we see if this component is an extender provider, and if so add it to the list we maintain. Yes, we have to write the functionality to keep track of those. All the magic is taken out of the designer architecture when you have to implement it yourself. Finally we add the component to our internal container instance.

```
public void Add(System.ComponentModel.IComponent component, string name)
{
        IDesigner designer = null;
        DesignSite site = null;

        // Check we're not trying to add a null component
        if (component == null)
                throw new ArgumentNullException("Cannot add a null component " +
                "to the container.");

        // Remove this component from its existing container, if applicable
        if (component.Site != null && component.Site.Container != this)
                component.Site.Container.Remove(component);

        // Make sure we have a name for the component
        if (name == null)
        {
                INameCreationService nameService = (INameCreationService)GetService(
                typeof(INameCreationService));
        name = nameService.CreateName(this, component.GetType());
        }

        // Make sure there isn't already a component with this name in the container
        if (ContainsName(name))
                throw new ArgumentException("A component with this name already " +
                "exists in the container.");

        // Give the new component a site
        site = new DesignSite(this, name);
        site.SetComponent(component);
        component.Site = site;

        // Let everyone know there's a component being added
        if (ComponentAdding != null)
                ComponentAdding(this, new ComponentEventArgs(component));

        // Get the designer for this component
        if (components.Count == 0)
        {
                // This is the first component being added and therefore
                // must offer a root designer
                designer = TypeDescriptor.CreateDesigner(component,
                typeof(IRootDesigner));
                rootComponent = component;
        }
        else
        {
                designer = TypeDescriptor.CreateDesigner(component,
                typeof(IDesigner));
        }

        // If we got a designer, initialize it
        if (designer != null)
        {
                designer.Initialize(component);
                designers[component] = designer;
        }
        else
        {
                // This should never happen
                component.Site = null;
                throw new InvalidOperationException("Failed to get designer for " +
                "this component.");
        }

        // Add to our list of extenderproviders if necessary
        if (component is IExtenderProvider)
        {
                IExtenderProviderService e = (IExtenderProviderService)GetService(
                typeof(IExtenderProviderService));
                e.AddExtenderProvider((IExtenderProvider)component);
        }

        // Finally we're able to add the component
        components.Add(component.Site.Name, component);
        if (ComponentAdded != null)
                ComponentAdded(this, new ComponentEventArgs(component));
}
```

IContainer.Remove

Although not quite as complicated as IContainer.Add, this method is responsible for the cleanup process of removing a component from the design surface.

The first couple of check we do are to make sure the component passed isn't null, and to make sure the component actually belongs to our design surface. Then we wrap the rest of the method in ComponentRemoving and then ComponentRemoved calls so that classes listening to our IComponentChangeService implementation know what's going on.

We have to remove the component from our list of extender providers if necessary, then dispose of and remove the designer associated with it. After that, we set the component's Site to null and that's all that's needed.

```
public void Remove(System.ComponentModel.IComponent component)
{
        ISite site = component.Site;
        IDesigner designer = null;

        // Make sure component isn't null
        if (component == null)
                return;

        // Make sure component is sited here
        if (component.Site == null || component.Site.Container != this)
                return;

        // Let the nice people know the component is being removed
        if (ComponentRemoving != null)
                ComponentRemoving(this, new ComponentEventArgs(component));

        // Remove extender provider (if any)
        if (component is IExtenderProvider)
        {
                IExtenderProviderService e = (IExtenderProviderService)GetService(
                typeof(IExtenderProviderService));
                e.RemoveExtenderProvider((IExtenderProvider)component);
        }

        // Remove the component and dispose of its designer
        components.Remove(site.Name);
        designer = (IDesigner)designers[component];
        if (designer != null)
        {
                designer.Dispose();
                designers.Remove(component);
        }

        // Let the nice people know the component has been removed
        if (ComponentRemoved != null)
                ComponentRemoved(this, new ComponentEventArgs(component));

        // Kill the component's site
        component.Site = null;
}
```

Implementing ISite

Our ISite implementation, called DesignSite, is going to be a poweful little beast that also implements IDictionaryService, the interface that for some reason some components use instead of using their own internal dictionaries. There isn't really much point going in to any of the stuff in this class in detail because it's mostly just template code except in the setter for the Name property.

The setter needs to check there isn't already a component with the name passed in the container. If there isn't, it can then proceed. To play friendly with the rest of the design environment, it must cause the IComponentChangeService implementation to raise the OnComponentChanging, OnComponentRename and OnComponentChanged events in that order while the change is made.

Implementing IComponentChangeService

This is an easy one. All this interface has is seven events and two methods which raise two of the events. All we do is add code for those two methods, and add another, internal method which is needed for our DesignSite class to raise the ComponentRename event. Although implementing this interface is easy, what's hard is working out where else in our implemented code to raise the events on it.

```
public void OnComponentChanged(object component, System.ComponentModel.
MemberDescriptor
member, object oldValue, object newValue)
{
        if (ComponentChanged != null)
                ComponentChanged(this, new ComponentChangedEventArgs(component,
                member, oldValue, newValue));
}

public void OnComponentChanging(object component,
System.ComponentModel.MemberDescriptor member)
{
        if (ComponentChanging != null)
                ComponentChanging(this, new ComponentChangingEventArgs(component,
member));
}

internal void OnComponentRename(object component, string oldName, string newName)
{
        if (ComponentRename != null)
                ComponentRename(this, new ComponentRenameEventArgs(component,
                oldName, newName));
}

public event System.ComponentModel.Design.ComponentEventHandler ComponentAdded;
public event System.ComponentModel.Design.ComponentEventHandler ComponentAdding;
public event System.ComponentModel.Design.ComponentChangedEventHandler
ComponentChanged;
public event System.ComponentModel.Design.ComponentChangingEventHandler
ComponentChanging;
public event System.ComponentModel.Design.ComponentEventHandler ComponentRemoved;
public event System.ComponentModel.Design.ComponentEventHandler ComponentRemoving;
public event System.ComponentModel.Design.ComponentRenameEventHandler ComponentRename;
```

Implementing IDesignerHost

This should really have been the class we started with, but IDesignerHost, IContainer, ISite and IComponentChangeService all depend on each other so much it makes sense to write about them in the order it's necessary to code them in. I'll cover the implementations of the members on IDesignerHost as I write them as best I can.

- *Activate* - This method is called to activate the root component. Our implementation will grab ISelectionService and set the root component as the primary selection.
- *CreateComponent* - This method is used by designers to have the host environment create an instance of a class that is to be situated on the design surface. Calling it is functionally identical to creating instance themselves then adding it to the container.
- *DestroyComponent* - This is called to remove a component from the design surface and dispose of it properly. This method is also responsible for using IComponentChangeService to let others know what is going on as the component is being removed.
- *CreateTransaction* - Creates an instance of our template DesignerTransaction class (I'll come on to that in a minute) and adds it to the stack of transactions.
- *GetDesigner* - We just use our designers hashtable to return the designer associated with the passed component.
- *GetType* - We check if there is an ITypeResolutionService available, and if so use it. If not, return Type.GetType() instead.
- *Container* - Just return the instance of the class since it implements IContainer itself.
- *InTransaction* - Return true if the number of transactions in our stack is greater than zero.
- *Loading* - For this example we are not concerned with loading hosts, so just return false.
- *RootComponent* - Return rootComponent, the first component added to our container.
- *RootComponentClassName* - Return the name of the class the first component added to our container was created from.
- *TransactionDescription* - Return the name of the transaction at the top of our stack, if there is one.

Designer Transactions

Every small change to a component should go through the IComponentChangeService, but when lots of small changes need to be wrapped up logically, that's where designer transactions come in. If we were writing a full-scale implementation of a design environment, we would keep track of these for undo/redo support.

There is no interface to be implemented here, we just need to inherit the abstract class DesignerTransaction with our own. All ours will do is make sure the appropriate events are raised by the host when the transaction is committed or cancelled.

Extender Services

The host environment needs to maintain a list of extender providers, and provide one of its own. You could probably get away without implementing these but they're pretty trivial and for completeness should definitely be included. We'll make a class called ExtenderServices which will implement IExtenderListService and IExtenderProviderService.

IExtenderListService has just one method, GetExtenderProviders. This is easily implemented with a simple ArrayList.

IExtenderProviderService exposes the same as IExtenderListService only it includes methods to add and remove the extenderproviders to and from the list. That's all we will put in this class. We instantiate it in the designer host's constructor and add its services to the servicecontainer along with the rest.

Our DesignerHost class has to implement IExtenderProvider itself. We need to do this because we want to have that (name) entry in the propertygrid, and for that, code has to be written. That said, there isn't much to it - we just want to extend all objects of type IComponent, with a new property "name" which is parenthesized. We have already written to code to wrap getting and setting component names in our DesignSite class, so the rest is simple.

```
internal class ExtenderServices : IExtenderListService, IExtenderProviderService
{
        ArrayList extenderProviders = new ArrayList();

        public ArrayList ExtenderProviders
        {
                get
                {
                        return extenderProviders;
                }
        }

        public ExtenderServices()
        {
        }

        public IExtenderProvider[] GetExtenderProviders()
        {
                IExtenderProvider[] e = new IExtenderProvider[extenderProviders.Count];
                extenderProviders.CopyTo(e, 0);
                return e;
        }

        public void RemoveExtenderProvider(System.ComponentModel.IExtenderProvider 
provider)
        {
                extenderProviders.Remove(provider);
        }

        public void AddExtenderProvider(System.ComponentModel.IExtenderProvider 
provider)
        {
                extenderProviders.Add(provider);
        }
}
```

Implementing ISelectionService

While this service is fairly simple in what it does, the implementation is important to get right. Selected components are kept track of internally with a simple ArrayList, and the only method of note is the SetSelectedComponents method. This method has to actually look at the state of the control and shift keys to cater for the various standard selecting operations.

This class also subscribes to the IComponentChangeService, because when a component is removed the selectionservice needs to know, if it is selected. Once the deleted component has been removed from the list of selected components, if there are none left the root component is selected.

The behaviour of this class is copied from that of Visual Studio so the shift and control keys work in the same way. Clicking on an already selected component makes it the primary selection.

Implementing ITypeDescriptorFilterService

This is an easy one. When the designers add/remove/shadow properties on the components they're designing, they do so because this method tells them to. When the propertygrid requests info on the members of the component, it queries for this interface on the component's ServiceProvider. If it finds it, it uses the simple members on the interface to alter any of the properties before they are displayed.

All we have do to when we implement this interface is (for each of the three methods) get the designer for the component being passed (easy, since we've already implemented the GetDesigner method on IDesignerHost) and call the designer's PreFilterProperties and PostFilterProperties (or equivalent) methods with the attributes we were passed.

Implementing INameCreationService

This service is implemented on a different level to the rest of the services we have added so far. This example is very simple so it will actually end up being added to the same servicecontainer as the rest, but let's consider the example of Visual Studio again.

The INameCreationService interface is used by the code we've written already to come up with a name for a component being added to the design container, if none was supplied. When you add a textbox to a form in Visual Studio, it gets the name "TextBox1" if you're writing in VB, and "textBox1" if you're writing in C#. That's because this service is implemented on a project level. Remember how servicecontainers are linked together in a tree? The request is made to the servicecontainer at the document view level but cascades up until it finds one.

It's a very simple interface to implement. You have the CreateName function, in which we will use the same naming algorithm as Visual Studio uses when you're writing in VB. We will just increment an integer counter until we find a name that isn't already in use.

The IsValidName function ensures that a name is valid. For this example we'll assume the name is going to be persisted to code at some point, and apply standard rules such as no spaces, and only alphanumeric characters are allowed. The ValidateName function just calls IsValidName, and if that returns false it throws an exception.

Implementing IUIService

This service is implemented only once and added to the top-level service container. It is how designers show messages, errors, popups and other windows. The documentation on this interface is pretty good so I won't explain what all the methods do here.

Implementing IToolboxService and IMenuCommandService

This is actually a fairly complicated interface to implement. It is the gateway between the toolbox user interface in the development environment and the designers. The designers constantly query the toolbox when the cursor is over over them to get feedback about the selected control.

Properly implementing IToolboxService is beyond the scope of this article so I will write only a skeleton implementation. It will be enough so that you can select a tool (or the pointer) and create and manipulate components on the design surface.

IMenuCommandService is looked for by some controls and components and they don't fail gracefully when it isn't found. This interface is responsible for managing standard menu commands, which the designers add to it. We will use it to execute the Delete command that has been added to it when the user hits the delete key. This is the correct way of letting the user delete components from the design surface.

Making it Work

That's about it for the DesignerHost and its related classes. The way we've written it, all it needs to be instantiated is a ServiceContainer instance. The framework provides a simple implementation of this, so we'll just use that. We want a very simple design surface so we'll just make a form with one side reserved for a propertygrid.

Now, how do we actually put our DesignerHost to use? It comes back to the Root Designer we discussed earlier. The key method is has is GetView, which returns a Control that can be added to any other form or control. This control is what you see when you look at a form in design view; it has a white background, and shows the form in design mode sitting in the top left.

Once we have created our DesignerHost, we (the host application) needs to subscribe to the SelectionChanged event of the ISelectionService interface that we already implemented. This is so that we can display the properties of the selected objects in our propertygrid.

We will use the following code to instantiate a form, put it in to design mode by adding it to our host, get its design view and add that view to the form:

```
private void Initialize()
{
        IDesignerHost host;
        Form form;
        IRootDesigner rootDesigner;
        Control view;

        // Initialise service container and designer host
        serviceContainer = new ServiceContainer();
        serviceContainer.AddService(typeof(INameCreationService),
        new NameCreationService());
        serviceContainer.AddService(typeof(IUIService), new UIService(this));
        host = new DesignerHost(serviceContainer);

        // Add toolbox service
        serviceContainer.AddService(typeof(IToolboxService), lstToolbox);
        lstToolbox.designPanel = pnlViewHost;
        PopulateToolbox(lstToolbox);

        // Add menu command service
        menuService = new MenuCommandService();
        serviceContainer.AddService(typeof(IMenuCommandService), menuService);

        // Start the designer host off with a Form to design
        form = (Form)host.CreateComponent(typeof(Form));
        form.TopLevel = false;
        form.Text = "Form1";

        // Get the root designer for the form and add its design view to this form
        rootDesigner = (IRootDesigner)host.GetDesigner(form);
        view = (Control)rootDesigner.GetView(ViewTechnology.WindowsForms);
        view.Dock = DockStyle.Fill;
        pnlViewHost.Controls.Add(view);

        // Subscribe to the selectionchanged event and activate the designer
        ISelectionService s = (ISelectionService)serviceContainer.GetService(
        typeof(ISelectionService));
        s.SelectionChanged += new EventHandler(OnSelectionChanged);
        host.Activate();
}
```

Conclusion

Reading over the code should fill in any gaps that I missed out while writing this. Hopefully this article will encourage more people to make use of the designer architecture in their own applications.

I have provided an example project that covers everything in this article.

Download C# Solution (95k)

Download VB Solution (19k) Ported by Ken Swinney

Revision History	
5 August 2003	Removed a small portion of code that originally came from SharpDevelop (GPL).
18 June 2003	It was pointed out to me that things like DrawGrid, SnapToGrid etc weren't appearing in the propertygrid for container controls like the Panel class. This was because I had forgotten to implement ITypeDescriptorFilterService, which I now have and is part of the article. I also noticed that I had forgotten to add the ProvidesProperty attribute to the DesignerHost, so the Name property which I'd gone to so much trouble to implement and make sure worked ok wasn't actually appearing in the propertybrowser either.