

Event-Based Programming Tutorial

Ted Faison
2009-05-04

This tutorial will introduce you to event-based programming (EBP) using a Windows desktop application written in C#. The application, called *SystemBrowser*, works a little like Windows Explorer: it displays folders and files, as shown in the next figure.

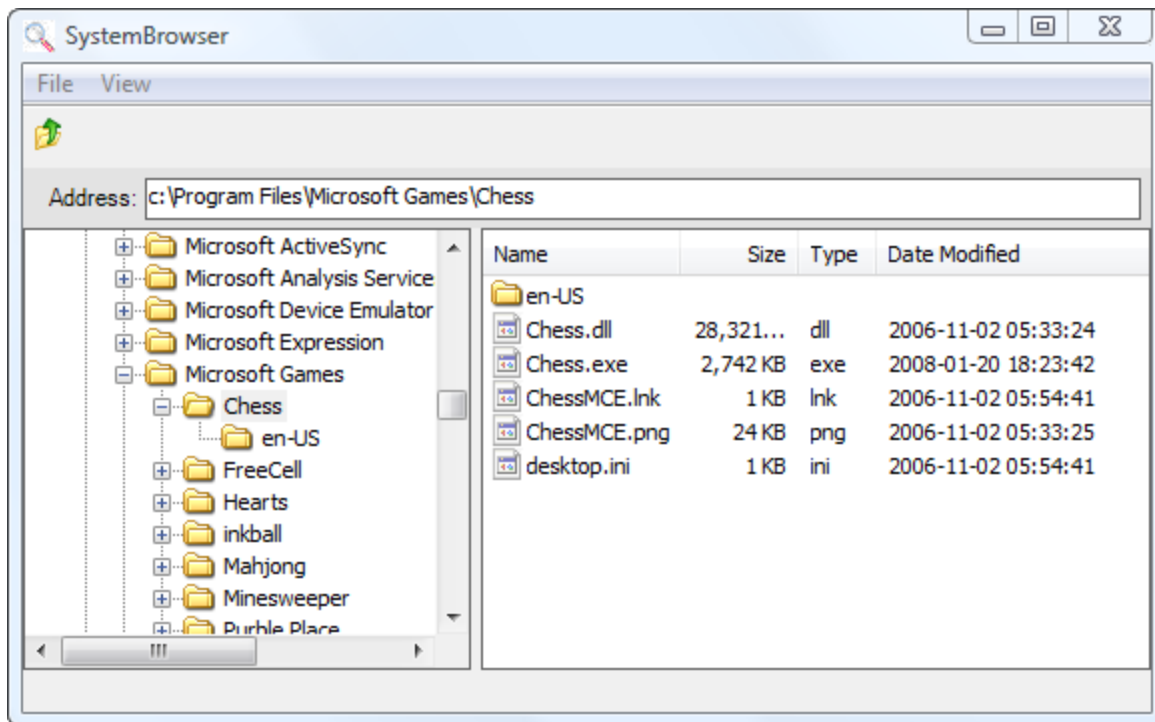


Figure 1 - The user interface of SystemBrowser.

What is Event-Based Programming?

Before diving into the design and implementation of SystemBrowser, let's back up for a second to discuss what EBP is, why it is appealing and how it allows us to control coupling. In a nutshell, EBP is a way to design a software system using events and event notifications to connect the major parts together. The advantage of this approach, over traditional *call-any-object-you-need* designs, is lower coupling. In many cases you eliminate static coupling between *parts*, making it possible to test those parts in isolation, using a test fixture.

I've barely begun and I've already introduced a couple of expressions that deserve explaining. What does *parts* mean in the previous sentence? I use this word to refer generically to classes, objects and assemblies. And what is *static coupling*. Unless you're read other material of mine discussing coupling, you're probably unfamiliar with the expression. Also, if there is something called *static coupling*, you might guess that there is something called *dynamic coupling*. You'd be right.

Static coupling is basically a dependency that impacts the build process. Say you have two classes, A and B. If A is statically coupled to B, then B must be present in order to build A. One way to introduce static coupling is for A to create an instance of B. The instantiation statement can't be compiled unless the compiler knows about, and therefore has access to, class B. In languages like C#, we make a class *known* to the compiler by including its parent assembly as a Reference in the project containing A. Obviously if A and B are in the same assembly, this won't be necessary, but A is nonetheless statically coupled to B. The convenience of having A and B packaged in the same assembly simply means that B will *always* be available when A is compiled.

Dynamic coupling is a dependency that affects runtime. If two parts A and B are dynamically coupled, it means that A can't be executed unless B is present at runtime.

Enough about coupling for the moment. I don't want to bore you with definitions before we even get started. Just remember this (and you probably already know it from experience): coupling creates problems, so it should be controlled and minimized where possible. It turns out that coupling is very controllable, if you design your software in an event-based manner. To demonstrate the advantages of event-based design, I'll show how to implement SystemBrowser the traditional, object-oriented way, with objects calling each other directly. As you will see, this will quickly get us into trouble, with circular coupling forcing us to resort to the introduction of interfaces to dig us out. After you have seen a traditional design, we'll revisit SystemBrowser using an event-based approach.

Requirements for SystemBrowser

Before we can design something, we need to know the requirements, so let's see what exactly what SystemBrowser needs to do. Like I said earlier, the program lets you browse the files and folders on your computer. For simplicity, SystemBrowser will only look at the C: drive. You can enter an explicit path in the address bar. When you hit the Enter key, the left pane will select the folder and the right pane will show the contents of the folder. If you hit the button on the toolbar (the one with the upward pointing arrow), SystemBrowser will navigate up one level to the parent folder, if one exists. The other parts of the system, including the address bar, the status bar, the left pane and the right pane, will all be updated with the new path. If the user double-clicks a folder in the right pane, SystemBrowser will navigate to that folder and again keep the others parts of the system updated.

The right pane has an additional behavior, controlled by the View menu. It can show its contents in either **Icon** mode or **Detail** mode, as seen in the following 2 figures.

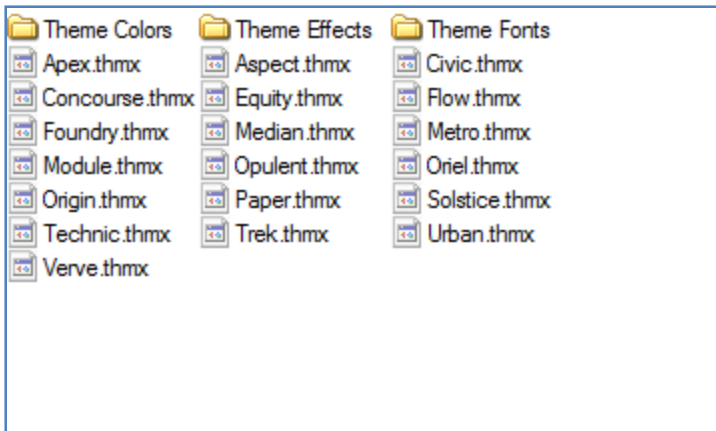


Figure 2- The right pane in *Icon* mode.

Name	Size	Type	Date Modified
Theme Colors			
Theme Effects			
Theme Fonts			
Apex.thmx	253 KB	thmx	2006-10-26 20:06:22
Aspect.thmx	66 KB	thmx	2006-10-26 20:06:22
Civic.thmx	99 KB	thmx	2006-10-26 20:06:24
Concourse.thmx	72 KB	thmx	2006-10-26 20:06:24
Equity.thmx	67 KB	thmx	2006-10-26 20:06:26
Flow.thmx	72 KB	thmx	2006-10-26 20:06:26
Foundry.thmx	62 KB	thmx	2006-10-26 20:06:26
Median.thmx	81 KB	thmx	2006-10-26 20:06:28

Figure 3- The right pane in *Detail* mode.

I've kept the requirements fairly simplistic, to avoid distracting you with unnecessary details.

A Traditional Design

Before looking at an event-based design, let's see how we might have designed SystemBrowser without using events. We'll start by listing the various classes we need. Just by looking at Figure 1 we can tell that we need the following, at a minimum:

- A main form to host the overall application.
- A `TreeView` for the left pane, showing a hierarchical list of folders.
- A `ListView` for the right pane, showing the folders and files in a given folder.

We could easily build the whole application using a single Form class. The class would contain a menu, a toolbar, a status bar, a `TreeView` and a `ListView`. No problem, but that class would get complicated pretty quickly as we added features.

In a typical project, with multiple programmers involved, it is better to split a project into smaller pieces. Not only are the pieces easier to deal with, but multiple people can work on different pieces simultaneously. We'll create the following classes:

- `FormMain`, to house the overall UI, with the menu, tool bar, address bar and status bar.
- `NavigatorFolders`, to house the `TreeView` of folders for the left pane.
- `ContentFileList`, to house the `ListView` of folders and files for the right pane.

The following figure shows the layout of the `FormMain`.

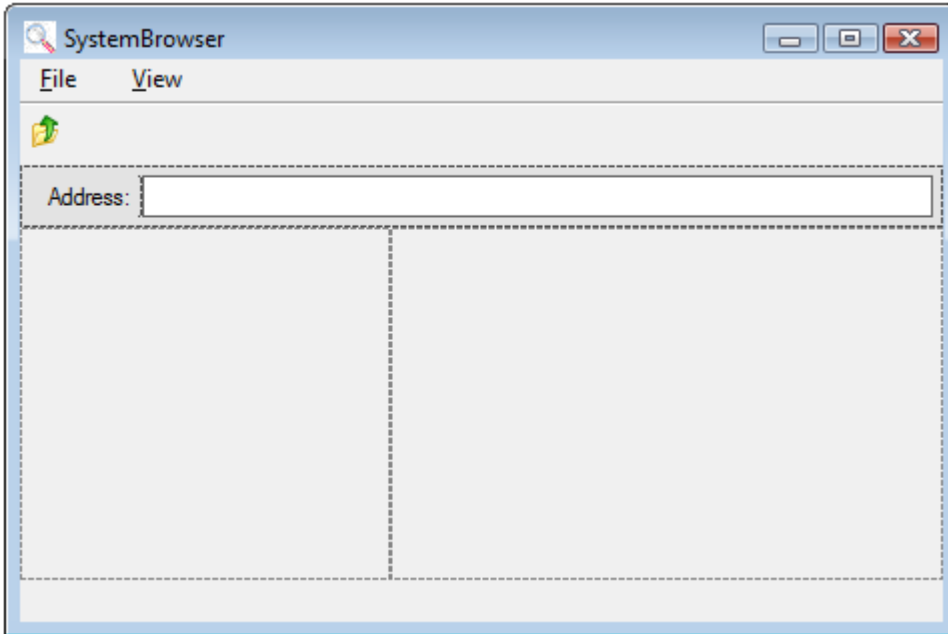


Figure 4 - The layout of `FormMain`.

The left and right panes are simple `UserControls` that contain a `TreeView` and a `ListView`, so there is little point in showing them in a figure. In the constructor for `FormMain`, we can instantiate `NavigatorFolders` and `ContentFileList` and add them to the UI, using code like this:

```
public FormMain()
{
    InitializeComponent();

    // create items
    contentFileList = new ContentFileList();
    navigatorFolders = new NavigatorFolders();

    // add item to left pane
    panelLeft.Controls.Add(navigatorFolders);
    navigatorFolders.Dock = DockStyle.Fill;

    // add item to right pane
    panelRight.Controls.Add(contentFileList);
    contentFileList.Dock = DockStyle.Fill;
}
```

Listing 1- A first-approach to creating `FormMain`.

`FormMain` uses two panels to host the `UserControls`. The implementation is pretty straightforward, right? Yes, but there are problems with the design. Consider how the left and right panes actually work. When the user clicks a folder in the left pane, we want to populate the right pane with the contents of the selected folder. This operation implies that `NavigatorFolders` calls `ContentFileList`. But we have an additional requirement: When the user double-clicks a folder in the right pane, we want the left pane to switch to that folder. This operation implies that `ContentFileList` calls `NavigatorFolder`. Now you can see the problem: there is circular coupling between `NavigatorFolders` and `ContentFileList`, because they need to call each other. Let's look at ways to address this type of problem, which is not unusual in software development.

Approach 1

We could have `FormMain` call the `NavigatorFolders` and `ContentFileList` constructors and pass references like this:

```
public FormMain()
{
    InitializeComponent();

    // create items
    contentFileList = new ContentFileList(navigatorFolders);
    navigatorFolders = new NavigatorFolders(contentFileList);

    // ...
}
```

Listing 2 - Approach 1 to the creation of the left and right pane objects.

The `ContentFileList` constructor would get a reference to the `NavigatorFolders` object and the `NavigatorFolders` constructor would get a reference to the `ContentFileList` object. This code would compile (with warnings) but not work. When calling the `ContentFileList` constructor, the `navigatorFolders` reference hasn't been initialized yet. When `ContentFileList` tries to use the reference, a null reference exception will occur. There are various tricks we could use to get around the problem, but we don't want to rely on tricks to make our code work. Remember, we want to keep things simple, and tricks go against the grain of simplicity.

Approach 2

If the first approach got us into trouble with null parameters, we could fix the problem by simply not using any parameters in the constructor calls. We could then pass to each created object a reference to the other object, by setting a property. The code would look like this:

```
public FormMain()
{
    InitializeComponent();

    contentFileList = new ContentFileList();
    navigatorFolders = new NavigatorFolders();

    contentFileList.NavigatorFolders = navigatorFolders;
    navigatorFolders.ContentFileList = contentFileList;

    // ...
}
```

Listing 3 – Approach 2 to the creation of the left and right pane objects.

This approach is fairly reasonable, so we'll use it. It isn't very elegant, but it works. But what if `ContentFileList` needed to talk to a large number of target objects? `FormMain` would wind up having to initialize a whole slew of properties, one for each target. `FormMain` would need to have detailed knowledge of all the targets required by `ContentFileList`. This knowledge represents a form of coupling called *logic coupling*.

Quick digression: Logic coupling occurs between two parts A and B when one contains assumptions about the other. Side effects are a form of logic coupling. Say B has a method that accomplishes some task, but also changes something about the state of the system. The latter is a side effect. If A calls this method and relies on the side effect, then A and B are logically coupled. If B were edited to change the side effect, A could easily break, but the compiler won't help you catch the problem. The logic coupling problem would only show up at runtime, and only if you actually ran the code affected. Logic coupling is bad, because our compiler and development tools can't track it. Also, logically coupled code is often the result of poor design and can often be eliminated through refactoring.

Getting back to our problem with `FormMain` having to set a potentially long list of properties on `NavigatorFolders` and `ContentFileList`, you can see that this approach bleeds requirements of `NavigatorFolders` and `ContentFileList` into `FormMain`. Not good.

Passing Parent References to Children

OK. We were able to come up for a way to get `NavigatorFolders` and `ContentFileList` `UserControls` to call each other, but we have an additional requirement. When either of the `UserControls` is used to select a folder, we want the status bar and the address bar to show the folder path. This requirement implies that both `NavigatorFolders` and `ContentFileList` call `FormMain`, which hosts the status bar and address bar. So we have another circular coupling problem, because `FormMain` calls the two `UserControls`, but those also need to call `FormMain`. This circular coupling is slightly different from the earlier one, because the coupling is between classes that are involved in a parent-child relationship.

Circularly coupled parent-child classes are quite common in traditional object-oriented systems. The typical solution has the parent passing a reference to itself to instantiated children. With `SystemBrowser`, the `FormMain` code would look something like this:

```
public FormMain()
{
    InitializeComponent();

    // pass reference to ourself to our children
    contentFileList = new ContentFileList(this);
    navigatorFolders = new NavigatorFolders(this);

    contentFileList.NavigatorFolders = navigatorFolders;
    navigatorFolders.ContentFileList = contentFileList;

    // ...
}
```

Listing 4 – Passing a parent reference to instantiated children.

This approach works fine, as long as the parent and children classes all live in the same assembly. If not, the circular coupling will prevent us from building the project. The design would then need to be refactored to use interfaces as a decoupling mechanism. The following diagram shows one way.

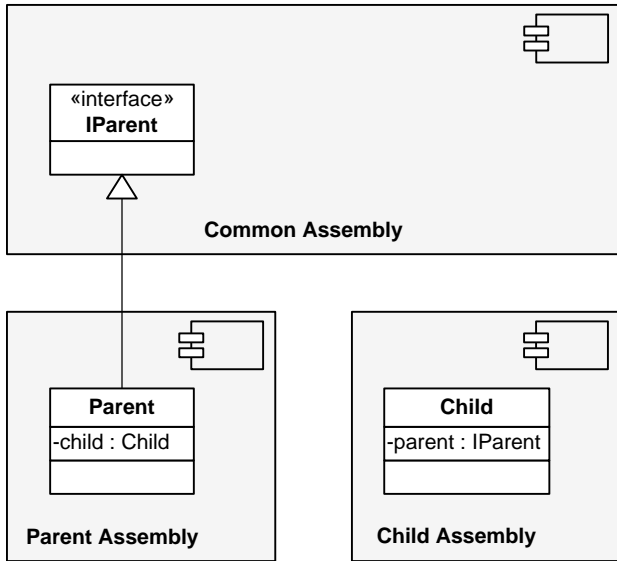


Figure 5 – Relaxing parent-child coupling using interfaces.

The `Parent` class implements the `IParent` interface, contained in the `Common` assembly, so the `Parent` assembly is statically coupled to the `Common` assembly. The `Child` class has an `IParent` reference, which also causes static coupling, but this time between the `Child` and `Common` assemblies. The coupling diagram looks like this:

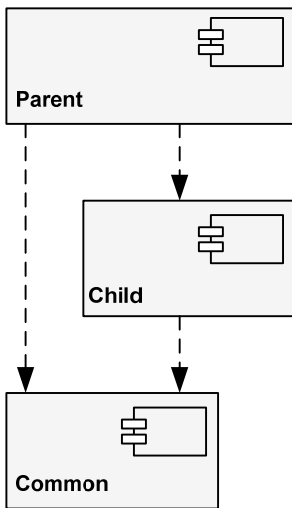


Figure 6 – Coupling diagram after introducing interfaces.

As you can see, there is no circular coupling anymore. Since the `Child` class contains only uses a generic `IParent` reference, the `Child` assembly only requires the presence of the `Common` assembly to compile. For the `Parent` assembly, things are a little different, because the `Parent` class creates an instance of the `Child` class like this:

```
public class Parent: Common.IParent
{
    Child child;
    public Parent()
    {
        child = new Child(this);
    }
}
```

```
}  
}
```

Listing 5 – How the Parent class creates instances of the Child class.

To create an instance of class `Child`, the Parent assembly must have a reference to the Child assembly. All is well with our design, but we had to do some reshuffling with interfaces to break the circular coupling.

An Event-Based Design

As should be obvious by now, one of the key goals of event-based design is to minimize coupling. And, for those with a short attention span, why do we want to minimize coupling? To make things simpler. Coupled parts are more complicated than decoupled ones, and they can't be used or tested in isolation. As a system gets larger, coupling tends to make complexity grow exponentially, instead of linearly.

Given the undesirable effects of coupling, we need to do everything we can to avoid introducing it into a system. One of the most common causes of static coupling is direct class instantiation. Anytime class A creates an instance of class B, A becomes statically coupled to B. The first priority in our design will therefore be to avoid having classes directly instantiate other classes, where feasible. A solution is to put all the class instantiation code in one place. To this end we'll create a special Builder class, described in the next section.

Centralizing Instantiations

Our first version of `SystemBrowser` had 3 main classes: `FormMain`, `NavigatorFolders` and `ContentFileList`. `FormMain` also internally includes a menu bar, a tool bar and a status bar. Let's create a Builder class that creates instances of everything:

```
public static class Builder  
{  
    public static Binder Binder;  
    public static FormMain FormMain;  
    public static FormMenuToolBar FormMenuToolBar;  
    public static StatusBar StatusBar;  
    public static NavigatorFolders NavigatorFolders;  
    public static ContentFileList ContentFileList;  
  
    public static void Build()  
    {  
        // create all UI elements  
        Binder = new Binder();  
        FormMain = new FormMain();  
        FormMenuToolBar = new FormMenuToolBar();  
        StatusBar = new StatusBar();  
        NavigatorFolders = new NavigatorFolders();  
        ContentFileList = new ContentFileList();  
    }  
}
```

Listing 6 – Using a Builder to create instances of all main classes.

Very simple, but what advantages are gained by using `Builder` to instantiate the main classes? The purposes of `Builder` are threefold:

- To be the sole point of instantiation of the application's main classes.
- To be absolutely clear about how and where those instances are kept in scope.

- To be the sole class coupled to the main classes in the system. Someone has to create those classes and that same someone is going to wind up statically coupled to them. The `Builder` is a good place to concentrate all the coupling, because the `Builder` is a near-trivial class, with essentially no business logic.

In traditionally-designed systems, it is not obvious which objects are keeping others in scope. If you use a `Builder` to create the main parts of your application, you'll never need to wonder who or what is keeping your objects alive (i.e. in scope, so they don't get garbage collected). Everything is referenced by the `Builder`, which acts pretty much as the anchor of the runtime object graph.

I created `Builder` as a static class, meaning that all its fields and methods are static. There can only be one instance of `Builder` in the system, so making the class static means not having to worry about instances. We could also have use a `Singleton` design pattern, to allow only a single instance of `Builder` to be created. The static class declaration turned out to yield a slightly simpler implementation. All the instances created in `Builder` are exposed directly as public fields. In a production system I would have exposed the fields as read-only properties.

Keeping FormMain Simple

Looking at the instances created in the `Builder`, there are two classes that need explanation: `Binder` and `FormMenuToolBar`. I'll discuss `Binder` a bit later. Class `FormMenuToolBar` was created as a `Form` to host the main menu bar, tool bar and address bar at design time. The following figure shows `FormMenuToolBar` in the Visual Studio Forms Designer.

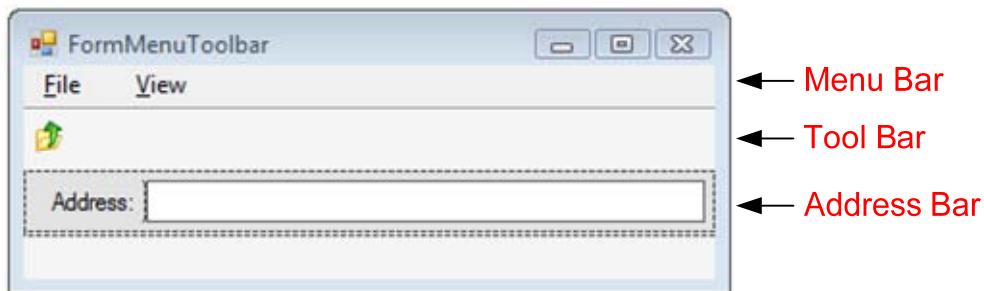


Figure 7 – The structure of `FormMenuToolBar`.

It makes sense to keep the menu and toolbar handlers in the same class (`FormMenuToolBar`) because in many cases a menu or toolbar button will share the same handler. For example you might have a `FileOpen` menu command and also a `FileOpen` toolbar button.

Why use a separate `Form` to hold items that will ultimately appear in `FormMain`? The reason is to keep `FormMain` as simple as possible. In most projects, the main form tends to become the center of all UI processing code, handling menus, accelerator keys, shortcuts, etc. The reason for this is that Visual Studio automatically places on the main form any handlers you create for UI elements. For example, if you double click a form's menu item, Visual Studio creates a handler for it in the form. If you double click the toolbar, Visual Studio again creates a handler in the form. Having the main form host all the UI event handlers is fine for simple applications, but as things get bigger and more complicated, you wind up having too much logic in a single class. This class quickly becomes a bottleneck in a large project, because a sizeable proportion of UI changes require modifying the main form. The main form of an application really should have only to tasks:

- To host the main UI elements, without controlling them or knowing what they do.
- To control the lifecycle of the application, by ending the application when the form is closed. The main form basically keeps the application itself in scope. When the form is closed, the application goes out of scope and terminates.

To host the app’s main UI elements doesn’t mean that `FormMain` actually controls or handles those elements. It just shows them on the screen. At runtime, right after building the menu and tool bar UI elements, we can move them from `FormMenuToolBar` into `FormMain`, leaving all the event handlers where they belong: in `FormMenuToolBar`. The following listing shows the Builder code:

```
public static class Builder
{
    public static void Build()
    {
        // create all UI elements
        Binder = new Binder();
        FormMain = new FormMain();
        FormMenuToolBar = new FormMenuToolBar();
        StatusBar = new StatusBar();
        NavigatorFolders = new NavigatorFolders();
        ContentFileList = new ContentFileList();

        // attach UI elements to FormMain
        FormMain.Menu = FormMenuToolBar.mainMenu;
        FormMain.Toolbar = FormMenuToolBar.panelToolBar;
        FormMain.Statusbar = StatusBar;
        FormMain.NavigatorFolders = NavigatorFolders;
        FormMain.ContentFolders = ContentFileList;
    }
}
```

Listing 7 – Merging UI elements at runtime into `FormMain`.

By having `FormMain` host, but not control, the UI, `FormMain` is almost trivial. It contains the panel structure for the UI, but has no functionality. The panel structure is shown in the next figure.

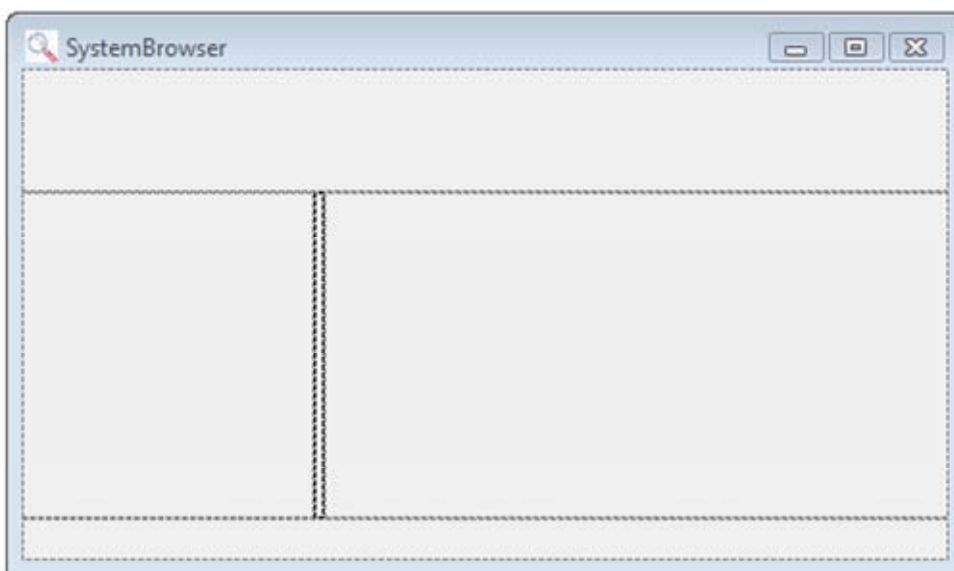


Figure 8 – The layout of `FormMain`.

The only code needed in `FormMain` is to support the hosting of the various UI children, such as the `NavigatorFolders` `UserControl`, the status bar, etc. Here is the code for `FormMain`:

```
public partial class FormMain : Form
{
    #region UI Children
    public Control Toolbar
    {
        set
        {
            value.Dock = DockStyle.Fill;
            panelToolBar.Controls.Add(value);
        }
    }

    public Control Statusbar
    {
        set
        {
            value.Dock = DockStyle.Fill;
            panelStatusBar.Controls.Add(value);
        }
    }

    public Control NavigatorFolders
    {
        set
        {
            value.Dock = DockStyle.Fill;
            panelLeft.Controls.Add(value);
        }
    }

    public Control ContentFolders
    {
        set
        {
            value.Dock = DockStyle.Fill;
            panelRight.Controls.Add(value);
        }
    }
    #endregion

    public FormMain()
    {
        InitializeComponent();
    }
}
```

Listing 8 – The complete code for `FormMain`.

It's hard to imagine how `FormMain` could be any simpler. The class has virtually no logic and has no references to the other classes in the system. Its sole purpose is to host UI controls. As far as `FormMain` goes, the other UI elements it hosts are plain `Control` instances.

Binding Event Sources to Event Handlers

So now we have instances of all the main classes. In order for the system to work, those instances will somehow have to talk to each other. If we allowed them to call each other directly, we would introduce static coupling between them, so direct communication is out of the question. What we want is for the instances to talk to each other indirectly, and we'll use event notifications. When one instance needs to notify others about something, it

will fire an event. The event source won't know who will be handling its events, and it shouldn't care. Its job is to notify the system about changes. How the system reacts to the changes is beyond the scope of the event source.

In our event-based design, classes act as event sources and event handlers. In traditional designs that use events, it is common for the event handling class to register itself as a target on the event source. Visual Studio uses this approach for UI elements. For example, if a Form contains a button with a Click handler, Visual Studio will automatically create the following code:

```
public class Form1 : Form
{
    Button button1;
    private void InitializeComponent()
    {
        // ...
        button1 = new Button();
        button1.Click += button1_Click)
    }

    private void button1_Click(object sender, EventArgs e)
    {
    }
}
```

Listing 9 – How Visual Studio creates a Click handler for a button.

Having the event handling class register itself as a target on the event source introduces coupling between the handler and the source. In the code above, this wasn't a problem because the handler class (`Form1`) was already statically coupled to the source class (`Button`). The coupling was due to `Form1`'s call of the `Button` constructor. In the general case of event sources and handlers, we don't want to introduce coupling between the two. This means we can't have one class instantiate the other or register itself as a handler for the other's events. We need to remove the event registration code from all classes and centralize it in a class. We'll call this class `Binder`. The `Binder` will wire up all the necessary events in one instance with handlers in other instances. All the wiring between the main instances in the system is now established in one place. The following listing shows what our `Binder` looks like:

```
public class Binder
{
    public void Bind()
    {
        Builder.FormMenuToolBar.ViewIcons += Builder.ContentFileList.ShowIcons;
        Builder.FormMenuToolBar.ViewDetails += Builder.ContentFileList.ShowDetails;
        Builder.FormMenuToolBar.UpClicked += Builder.NavigatorFolders.SelectParentFolder;
        Builder.FormMenuToolBar.AddressChanged += Builder.NavigatorFolders.SelectFolder;

        Builder.NavigatorFolders.FolderChanged += Builder.ContentFileList.Populate;
        Builder.NavigatorFolders.FolderChanged += Builder.FormMenuToolBar.ShowAddress;
        Builder.NavigatorFolders.Message += Builder.StatusBar.ShowMessage;

        Builder.ContentFileList.Message += Builder.StatusBar.ShowMessage;
        Builder.ContentFileList.FolderDoubleClicked += Builder.NavigatorFolders.SelectFolder;
    }
}
```

Listing 10 – The Binder for SystemBrowser.

The `Binder` wires events to handlers. I haven't described the events, but just by looking at the listing you can understand pretty much what is going on. As an example, let's take the `FolderChanged` event exposed by

`NavigatorFolders`. Without knowing much about how the system is implemented, you can probably guess that the `FolderChanged` event is fired when the user changes the selected folder in the folder list. When this happens, we need to show the list of files and folders in the right pane. We do so by having the `ContentFileList.Populate` method handle the event. You can probably guess that this method fills the `ListView` in the `UserControl` with folder and file entries. When a folder change occurs in `NavigatorFolders`, we also want the address bar to show the newly selected folder. We do so by having `FormMenuToolBar.ShowAddress` method handle the event. In this example, the `FolderChanged` event has two handlers, but an event can usually have any number of handlers, including zero.

Circular coupling is not a problem in our event-based design. Instances can call each other (via event notifications) without any problem, because there is absolutely no static coupling between event sources and event handlers. For example, looking at the previous listing, you can see that the `FolderChanged` event of `NavigatorFolders` is handled by `ContentFileList`. The `FolderDoubleClicked` event of `ContentFileList` is handled by `NavigatorFolders`. Neither class is aware of, or coupled to, the other.

Starting the Program

Every program needs some sort of initialization and `SystemBrowser` is no exception. Who tells the `Builder` to instantiate all the classes? Who tells the `Binder` to wire up the system? How do we tell the main form what to display when the program starts? One solution might be to use the `Program` class that Visual Studio automatically creates automatically. Before starting the application's message pump, we could use code like this to set everything up:

```
static class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles(); // enable XP and Vista GUI styles
        InitializeSystem();
        Application.Run(Builder.FormMain);
    }

    static void InitializeSystem()
    {
        Builder.Build(); // instantiate all the top-level classes
        Builder.Binder.Bind(); // wire all the top-level objects

        // show initial folders and files
        const string initialFolder = @"c:\";
        Builder.NavigatorFolders.Populate(initialFolder);
        Builder.ContentFileList.Populate(initialFolder);
        Builder.StatusBar.ShowMessage(initialFolder);
        Builder.FormMenuToolBar.ShowAddress(initialFolder);
    }
}
```

Listing 11 - One way to initialize `SystemBrowser`.

In an event-based system, it should be more natural to use events to accomplish most tasks, including this simple initialization one. Look at the system binding, shown back in Listing 10, for a minute. We already have event wiring in place to make the system respond to `FolderChanged` events fired by `NavigatorFolders`. Let's

see if we can leverage this event for our purposes. We're in luck (luck has nothing to do with it!). All we need to do is handle the `Load` event in `NavigatorFolders` like this:

```
private void NavigatorFolders_Load(object sender, EventArgs e)
{
    string initialFolder = @"c:\";
    Populate(initialFolder);
    FireFolderChanged(initialFolder);
}
```

Listing 12 – A better and simpler way to initialize `SystemBrowser`.

That's it. System initialized. Of course we still need to call `Builder.Build` and `Binder.Bind`, in the `Program` class in Listing 11. It takes a bit of practice to start thinking in terms of events, but once you get the hang of it, you'll discover a new world of simplicity.

I won't be discussing the other details about the `NavigatorFolders` and `ContentFileList`, because those details have little to do with events. Most of the code is devoted to populating `TreeView`s and `ListView`s. Check out the source code for complete details.

Conclusion

That wraps up my tutorial. Bear in mind that `SystemBrowser` is a very simple program, so issues related to coupling and complexity are relatively small, regardless of whether you use an event-based design or not. But as you get into larger and larger systems, coupling and complexity quickly begin to dominate the landscape of problems. Testing becomes increasingly difficult. Some development methodologies, such as Test-Driven Development, recognize this and put a great deal of emphasis on testability as a design target. Event-based designs are eminently testable, because the parts to test can be used by themselves, outside the system they were designed to be part of. Another nice thing about EBP is that it can be applied to any type of software system, be it a Windows service, a Linux backend process, a Web application, a desktop application, a real-time system, whatever. My hope is to have piqued your curiosity about EBP enough so that it finds a way into your next project. There are a lot of other interesting aspects of EBP, such as these:

- Using a role-based approach (with things like Coordinators, Workers and Coordinator teams) to design event-based systems.
- Documenting event-based systems using Signal Wiring Diagrams.
- Getting a deeper understanding of coupling and how to avoid it.

For more information about these and other topics, see my book *Event-Based Programming: Taking Events to the Limit*, published by Apress.