Client Logic for MSIE 1. 0 Component Specification

1. Design

This component provides the client side logic needed to transform Internet Explorer into an Orpheus client by integrating the client logic into an Internet Explorer extension.

The client side logic is implemented in this component in the *MsieClientLogic* class and is completely decoupled from the mechanism to create an Internet Explorer extension.

This component also provides a generic mechanism for creating Internet Explorer extensions with the *WebBrowserSite* and *ToolBand* classes.

1.1 Design Patterns

IExtensionEventHandlerFactory is an **abstract factory** (strategy) pattern. It provides an interface for creating the custom *IExtensionEventHandler* instances.
Different **strategies** for IEventsManager, *IWebBrowserWindowNavigator* and *IPersistence* can also be plugged in.

The *ExtensionEventHandlerDelegate* wraps the **listener** methods to be invoked when an event is fired.

1.2 Industry Standards

COM, DOM, URI, HTML, JavaScript

1.3 Required Algorithms

1.3.1 Overview

This component as the requirements asks for, provides the client side logic needed to transform Internet Explorer into an Orpheus client by integrating the client logic into an Internet Explorer extension.

The client side logic is implemented in this component in the *MsieClientLogic* class and is completely decoupled from the mechanism to create an Internet Explorer extension.

To create an extension for the Internet Explorer several interfaces are exposed and need to be implemented by the extension. The extension must also be registered as a COM component, and a value must be added in the registry, in order for Internet Explorer to be aware of the extension.

This component provides a generic mechanism for creating Internet Explorer extensions with the *WebBrowserSite* and *ToolBand* classes. Clients will need to extend the *ToolBand* class in order to create the user interface for the extension, which by extending from *ToolBand* will take the form of an Internet Explorer tool band.

The details of how to integrate the client side login into the extension are presented in the demo section of this document.

1.3.2 How it works

Whenever an Internet Explorer window is opened, regardless of how it was opened, it will look in the registry for any extensions and will create a new extension every time. So, for every opened web browser window there will be an associated extension instance. The extension instance is a *ToolBand* derived class.

To have all these extension instances work with the same client logic instance, the client logic is created as a singleton by the extensions and as a result browser windows opened by the main browser window will share the same client logic singleton instance.

By sharing the same client logic instance, all the extension instances will have the ability to customize a new web browser window with the same object. Customizing the web browser implies providing a *IDocHostUIHandler* interface to the web browser. Through this interface, the web browser can query things like the scripting object or context menu to display. Internet Explorer provides the *ICustomDoc* interface so you can pass Internet Explorer your *IDocHostUIHandler* implementation. The *GetExternal* method of this interface returns to the web browser an object that it will make available to JavaScript code. The object's properties and methods will then be available to any page displayed in the browser through the document's external object.

1.3.3 Client Side Logic

The client side logic is implemented in the *MsieClientLogic* class. This class encapsulates all the required functionality needed to behave as an Orpheus application client.

It exposes an events manager through which event handlers can be specified either through the configuration file, or at development time.

A persistence mechanism is used to store values between browser restarts in the registry for every user.

The navigation in the main browser window or a new window is abstracted in the *IWebBrowserWindowNavigator*. The implementation provided reuses a popup browser window, which it opens by accessing the main browser DOM using *window.open*.

1.3.4 Internet Explorer Extensions

Browser extensions allow developers to provide easy access to their browser enhancements by adding elements to the default user interface. Beginning with Internet Explorer 5, this feature allows developers to add entries into the Tools menu and buttons to the toolbar.

Explorer extensions, explorer bars, tool bands, or desk bands although they can be used much like normal windows are COM objects that exist within a container. Explorer Bars are contained by Internet Explorer, and desk bands are contained by the Shell. While they serve different functions, their basic implementation is very similar. The primary difference is in how the band object is registered, which in turn controls the type of object and its container.

All band objects must implement *IDeskBand*, *IObjectWithSite* and *IPersistStream* interfaces. In addition to registering their class identifier (CLSID), the Explorer Bar and desk band objects must also be registered for the appropriate component category. Registering the component category determines the object type and its container.

For the tool bands to be integrated into Internet Explorer information must be added to the registry. Tool bands must have their object's CLSID registered with Internet Explorer. To do this, assign a value under **HKEY_LOCAL_MACHINE\Software\Microsoft\Internet Explorer\Toolbar** named with the tool band object's CLSID GUID.

More details on creating browser extensions can be found here:
http://msdn.microsoft.com/library/default.asp?url=/workshop/browser/ext/extensions.asp

1.3.5 HTTP Request Event Handler

These event handlers should be configured for user interface sharing the same common functionality to make a web request to a configured URL and display the result in the browser window, or in a reusable popup window.

1. Reads from the configuration file and based on the event name the URL of the page to request (the <event name>_url property).
2. Reads from the configuration file and based on the event name whether the page should be displayed in a new window (the <event name>_new_window property).
3. Reads from the configuration file and based on the event name the method to use for the request (the <event name>_method property).
4. Creates a new web request (WebRequest.Create) for the configured URL.
5. Sets the method to use to the request to the configured value.
6. Gets the response stream of the request.
7. It then uses the web browser window navigator from the context object to set the content to the browser.
8. Closes the response stream.

1.3.6 Displayed Page Changed Event Handler

Below are the steps taken when a new page is displayed in the browser.

1. Get the URL of the current page: Gets the browser document from the browser and then the location IHTMLLocation location = document.GetLocation(); Gets the text URL: string url = location.GetHref
2. Creates a new URI from the URL and gets the host name.
3. Test the Bloom Filter for the host name. If found then:
   3.1. Read from the configuration file the configured URL(document_completed_url property), set the host parameter using string.Format.
   3.2. Create a new web request for the created URL. Get the response stream, the result stream should be interpreted as number. If the number is not 0 then
      3.2.1. Read from the configuration file the configured URL(document_completed_0_responce_url property), set the host parameter and direct the web browser window navigator to the new location to open in a new window.

1.3.7 Polling for Updates Event Handler

The steps to follow when handling the poll for updates event:

1. Reads from the configuration file the configured URL(polling_url property).
2. Creates a new web request (WebRequest.Create) to the configured URL.
3. Gets from the persistence the “timestamp” value.
4. The URL is constructed with the update timestamp of the feed (string.Format(url, timestamp))
5. Gets the response stream from the request.
6. Parses the response stream using the RSS parser into a *RSSFeed* document.
7. For each *RSSItem* in the feed items:
   7.1. If the content is of type text, HTML or XHTML sets the content to be displayed in the new window using the web browser window navigator.
   7.2. If the content is of type “application/x-tc-bloom-filter” restore the bloom filter from the serialized content of the feed item.
8. Persist the feed timestamp.
9. Close the response stream.

1.3.8 Test Object Handler

The handler will use the *ExtensionEventArgs params* array to get an *IHtmlElement* object passed to the handler.

1. Gets the *IHTMLElement* for the *parameters* array.
2. Get the child elements of the element and for each one:
   2.1. Get the inner text trim the string, remove (blank, tab, CR, and LFCharacters) and normalize all sequences of white space to a single space, and folding all characters to lower case.
   2.2. Append all the elements text and use the hash algorithm to generate the SHA-1 hash of the UTF-8 encoding of the Unicode code point sequence.
3. Get from the persistence the stored hash. Compare to this hash and if they match:
   3.1. Read from the configuration file the “test_object_url” property
   3.2. The current game ID, the domain, the target sequence number, and the normalized and UTF-8 encoded text from which the hash was computed will be specified in the request URL as query parameters. (string.Format)
   3.3. Create a web request (WebRequest.Create) and get the response stream.
   3.4. Use the web browser window navigator from the context to show the response content.

1.3.9 Generating the Remote Callable Wrappers

The Remote Callable Wrapper types can be generated using IDL files.

The types required are defined in the mshtmhst.h header file and the mshtmhst.idl IDL file and the ocidl.h header file and ocidl.idl IDL file from the Platform SDK.

The IDL files will be used to generate the managed proxies for the required types.

To generate the wrapper for the *IObjectWithSite* interface the IDL file should be created as:

```
[
    uuid(854B9347-7147-40ce-9DE7-1DBCD761D4CE)
]
library Orpheus.Plugin.InternetExplorer.Interop
{
    import "OCIdl.Idl";

    interface IObjectWithSite;
};
```

All the other required types should be added to this file.

To generate the type library:

midl `Orpheus`.idl /tlb `Orpheus`.tlb

To convert the COM type library into common language assembly:

tlbimp `Orpheus`.tlb /out: `Orpheus.Plugin.InternetExplorer.Interop`.dll

1.4 Component Class Overview

WebBrowserSite

This class is the base class of the Internet Explorer extension model and provides for derived classes a *site* within the browser. In general, a *site* is an intermediate object placed in the middle of the container and each contained object.
This class implements *IObjectWithSite* interface so it's container can supply it with an interface pointer for its site object. Then, this class, can communicate directly with its site. A container can pass the *IUnknown* pointer of its site to an object through *IObjectWithSite.SetSite*. Callers can also retrieve the latest site passed to *IObjectWithSite.SetSite* through *IObjectWithSite.GetSite*. This usage provides a hooking mechanism, allowing third parties to intercept calls from an object to its container site object.
This class extends the Control class to allow derived classes to build their user interface.

Together with the *ToolBand* class, both classes provide a generic mechanism for creating Internet Explorer extensions. These classes are decoupled for the intended purpose of this component to provide the client side logic for a web application. The usage scenario is to derive the *ToolBand* class and there hook the client logic functionality implemented in the *MsieClientLogic* class.

ToolBand

This class extends the *WebBrowserSite* and implements the *IDeskBand* interface in order to provide a base class for all tool band objects. The *IDeskBand* interface is used by the browser to retrieve information about a band object.

MsieClientLogic

This class implements the client-side logic for interacting with the web application. This client logic will be incorporated into an Internet Explorer extension that enables Internet Explorer to be used as a client.

This class is decoupled from the Internet Explorer extension mechanism, to give clients a great level of flexibility when incorporating this logic into an Internet Explorer extension.

It is responsible with the creation of the logic objects through the configuration file and using the Object Factory component.

It hooks to web browser events in order to invoke component event handlers when a new page is displayed in the browser. It also starts a timer and invokes event handlers on regular time intervals.

This class can be used as a singleton, as well. The reason for using this class as a singleton is to have the same instance of this class for multiple opened browser windows. The way this works is as follows: when Internet Explorer is started it looks in the registry for an extension object and creates it using its GUID. In this case the extension will be some derived class of *ToolBand* class. So every web browser window, regardless how it was opened, will have a different instance of a derived *ToolBand* class. In order for web browser windows, opened by the main web browser like for example using *window.open*, to have a reference to the same *MsieClientLogic* object they must use it as a singleton.

One more requirement for the clients is to invoke the *CustomizeWebBrowser* method whenever a new window is opened and a new extension is created, passing the newly obtain reference of the web browser, in order to provide the browser with the same customization like scripting object or custom context menu.

DefaultDocHostUIHandler

This class implements the *IDocHostUIHandler* interface and provides the means to extend the Internet Explorer Document Object Model (DOM) with objects, methods, and properties.

This is done by providing MSHTML a pointer to the *IDispatch* interface for the COM automation object that implements the custom object (*ScriptingObject* class) properties, and methods. These objects, properties, and methods will then be available to any page displayed by the web browser through the document's external object.

Instances of this class are created by the *MsieClientLogic* class and are set to the browser. The only requirement on *IDocHostUIHandler* implementations is to provide a constructor with a (*MsieClientLogic*) parameter

There is only one method that this class actually implements, the *GetExternal* method which will set the pointer to the scripting object created by this class.

ScriptingObject

This class represents the object that will be exposed to page JavaScript code.

The browser will invoke on the *IDocHostUIHandler* the *GetExternal* method which will provide to the browser the scripting object, which web pages will be able to access through the external object:

```
function testScripting()
{
        //get the scripting object interface
        var scriptingObject = window.external;
        //invoke it
        scriptingObject.LoggedIn();
}
```

This class provides all the methods required to be accessible from JavaScript.

Scripting objects are created by the *MsieClientLogic* which uses the Object Factory to create the object using a (*MsieClientLogic*) constructor. Scripting objects must be set the Com visible attribute set to true.

This class should be marked with this attribute: [ComVisible(true)].

IExtensionEventsManager

This interface defines the contract that is required for any extension events manager to implement. An events manager is responsible for managing the delegates for every specific event identified by a name and to invoke them when that specific event is fired.

ExtensionEventArgs

Event arguments class for event handlers.

This class holds a reference to the context object, and provides handlers with the event name as well, for which they were invoked. As extra flexibility it also allows clients to pass to event handlers any object through the parameters array. The handlers will need to know of any of these objects if they are to use it.

IWebBrowserWindowNavigator

This interface defines the contract that is required for any browser window navigator to implement. A browser window navigator is responsible for displaying a new page to the user either inside the browser or in a new window.

This component provides an implementation of this interface that reuses an opened popup window to display the web pages.

IPersistence

This interface defines the contract that is required for any persistence mechanism to implement. A persistence mechanism is responsible for saving and retrieving simple values for a specified key.

This component provides an implementation of this interface that stores per user the values in the registry.

DefaultExtensionEventsManager

This class is the default implementation of the *IExtensionEventsManager* interface.

This class restores using a handler factory, the event handlers, creates a delegate for each handler and stores them, to be invoked when an event is fired.

IExtensionEventHandlerFactory

This interface defines the contract that is required for any event handler factory to implement. A event handler factory is responsible for creating the *IExtensionEventHandler* implementation based on a specified event name. The event name is simply used here to uniquely identify the required handler(s) implementation that needs to be created.

This component provides one implementation of this interface which uses the name to get from the configuration file the key, and the Object Factory component to create, using the key, the specified implementation(s).

IExtensionEventHandler

This interface defines the contract that is required for any event handler to implement. An event handler will be given a context object which holds all the details of the current extension.

For added flexibility to the design, a factory is used to create implementations of this interface based on a event name.

This component provides four implementations of this interface. Each one is responsible for handling a different type of event, all sharing the same mechanism to be created and invoked.

DefaultExtensionEventHandlerFactory

This class is an implementation of the *IExtensionEventHandlerFactory* interface.

This implementation will use the specified event name to get from the configuration file a value representing a key. This key will be used by the Object Factory component to create the required *IExtensionEventHandler* implementation. By using Object Factory component, constructor parameter values can be specified for the created implementation. The details of how to configure the Object Factory component can be found in the component’s specification document.

HttpRequestUserInterfaceEventHandler

This class is an implementation of the *IExtensionEventHandler* interface.

This event handler should be configured in the configuration file for user interface generated events, such as login button pressed. Most of these events require a HTTP

request to be made to a URL and the result displayed in either the main browser window or a popup window.

This class constructs the URL of the page to request and then directs the browser navigator to the specified URL.

WebBrowserDocumentCompletedEventHandler

This class is an implementation of the *IExtensionEventHandler* interface.

It performs a request to the server, if needed and gets a numeric value based on which it will direct the browser to a new page.

PollingEventHandler

This class is an implementation of the *IExtensionEventHandler* interface.

This event handler will be invoked at regular intervals or through the scripting method.

TestObjectEventHandler

This class is an implementation of the *IExtensionEventHandler* interface.

RegistryPersistence

This class is an implementation of the *IPersistence* interface.

It used Registry to save the values.

DefaultWebBrowserWindowManager

This class is an implementation of the *IWebBrowserNavigator* interface.

It will be responsible for displaying the requested page or the provided content inside either the main browser window or inside a opened and reused window.

The new opened window will have the same customization object set to it as the main browser window. When the new window is opened, the Internet Explorer will load the extension object again. The extension object, if used as depicted in the component specifications, will get the same instance of the *MsieClientLogic* by using the singleton of the class, and will call the CustomizeWebBrowser method which will set the same customization object to the newly opened browser.

ExtensionInstaller

Installer class for Internet Explorer Extensions.

Extensions must be set with the *ExtensionAttribute* in order to be recognized by the installer.

ExtensionAttribute

Extension attribute class. Tool band objects that need to be installed must be set with this attribute.

1.5 Component Exception Definitions

ClientLogicExtensionException

Base exception for the exceptions thrown by this component.

ConfigurationException

Exception to signal any problems with the configuration file and the Object Factory.

SiteSettingException

Exception to signal any problems when setting the browser site and getting a reference to the host browser.

EventHandlerCreationException

Exception to signal any problems when creating the event handlers for a event.

HandleEventException
Exception to signal any problems when handling an event.

WebBrowserCustomizationException

Exception to signal any problems when customizing the browser by providing a IDocHostUIHandler.

FireEventException

Exception to signal any problems while firing and handling and event.

PersistenceException

Exception to signal any problems with the persistence.

WebBrowserNavigationException

Exception to signal any problems with the browser navigator when setting a new location.

1.6 Thread Safety

The Internet Explorer extension classes that make up the base for the extension user interface do not need to be thread safe. These classes are **WebBrowserSite** and **ToolBand**.

The main class, the **MsieClientLogic** class will be used as a singleton for this reason the class and the aggregated objects should be thread safe.
There is also the scenario in which code can be run by multiple threads, when event handlers are invoked from different places at the same time, like from the browser window JavaScript code through the scripting object and from code. For this reason the classes involved in this scenario are expected to be thread safe.

The classes that present thread safety issues are **MsieClientLogic, DefaultWebBrowserWindowManager, DefaultExtensionEventsManager.** All these classes use 'lock' in order to be thread safe.

All the other classes involved in the client side logic are also thread safe by having no state or being immutable.

The used **Bloom Filter** component is not thread safe. The **MsieCientLogic** class will provide synchronization by locking.

The **RSS Library** component is not thread safe. This component uses the *Atom10Parser* to parse content into an RSS object model, which is thread safe and should create a new instance of the object model on every request, thus every thread will have different instances.

The **Hashing Utility** component is thread safe.

Exception classes have no state and are thread safe.

2. Environment Requirements

2.1 Environment

.Net Framework 1.1

2.2 TopCoder Software Components

- **Object Factory 1.0** for creating custom *IExtensionEventHandler* instances, the *IExtensionEventHandlerFactory* instance, the *IDocHostUIHandler, the IPersistence, the IExtensioEventsManager* implementations and the *ScriptingObject.*
- **Configuration Manager 2.0** for reading in configuration values.

- **Hashing Utility 1.0** to get the SHA1 hash for a value.
- **Bloom Filter 1.0** maintains the domains in the game.
- **RSS Library 1.1 to** parse feeds received from server and get the items.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

Orpheus.Plugin.InternetExplorer.Interop - Runtime Callable Wrappers for the needed types
Orpheus.Plugin.InternetExplorer - main implementation package.
Orpheus.Plugin.InternetExplorer.EventsManagers – the events manager implementations.
Orpheus.Plugin.InternetExplorer.EventsManagers.Factories - event handler factories
Orpheus.Plugin.InternetExplorer.Persistence – persistence implementations
Orpheus.Plugin.InternetExplorer.WindowNavigators – browser window navigators.

3.2 Configuration Parameters

Orpheus.Plugin.InternetExplorer.MsieClientLogic

Parameter	Description	Values
bloom_filter	The Object Factory key where the BloomFilter is configured.	String. Required.
window_navigator	The Object Factory key where the *IWebBrowserWindowNavigator* implementation to use is configured.	String. Required.
persistence	The Object Factory key where the *IPersistence* implementation to use is configured.	String. Required.
extension_events_manager	The Object Factory key where the *IExtensionEventsManager* implementation to use is configured.	String. Required.
doc_host_ui_handler	The Object Factory key where the *IDocHostUIHandler* implementation to use is configured.	String. Required.
scripting_object	The Object Factory key where the *scripting object* to create is configured.	String. Required.
poll_interval	The interval in minutes to poll for updates.	Numeric. Required

Orpheus.Plugin.InternetExplorer.EventsManagers.DefaultExtensionEventsManager

Parameter	Description	Values
event_handlers_factory	The Object Factory key where the *IExtensionEventHandlerFactory* implementation to use is configured.	String. Required.
events	Represents a list of events. Based on the event name the configured handlers are retrieved. Event names (ex: *DocumentCompleted*, *LogIn*, *LogOut*) should be added here.	Strings. Required.

Orpheus.Plugin.InternetExplorer.EventsManagers.Factories.

DefaultExtensionEventHandlerFactory

Parameter	Description	Values
<event name>_handlers	The name of the handlers configured to be invoked for the event. For each handler name there is another property: <handler name> _handler. Any number of handlers can be added here.	Strings. Required.
<handler name>_handler	The Object Factory key where the *IExtensionEventHandler* implementation to use is configured.	String. Required.

Orpheus.Plugin.InternetExplorer.EventsManagers.Handlers.

HttpRequestUserInterfaceEventHandler

Parameter	Description	Values
<event name>_url	The URL of the page to request as response to the event.	String. Required.
<event name>_new_window	Whether to display the result of the web request in a new window or not.	True or fale. Optional. Default=false.
<event name>_method	The method to use for the request.	GET or POST. Optional. Default=Get.

Orpheus.Plugin.InternetExplorer.EventsManagers.Handlers.
WebBrowserDocumentCompletedEventHandler

Parameter	Description	Values
document_completed_url	The url to make the request to. The host parameter will be placed in string as {0}.	String. Required. Ex:www.tc.com/? host={0}
document_completed_0_responce_url	The url to make the request to. The host parameter will be placed in string as {0}.	String. Required. Ex:www.tc.com/? host={0}

Orpheus.Plugin.InternetExplorer.EventsManagers.Handlers.PoolingEventHandler

Parameter	Description	Values
rss_parser	The Object Factory key of the IRSSParser implementation to use. Atom1.0 parser should be configured.	String. Required.
polling _url	The polling url.The timestamp parameter will be placed in string as {0}.	String. Required. Ex:www.tc.com/? timestamp={0}

Orpheus.Plugin.InternetExplorer.EventsManagers.Handlers.TestObjectEventHandler

Parameter	Description	Values
hash_algorithm	The Object Factory key of the HashAlgorithm implementation to use. The SHA-1 should be configured.	String. Required.
test_object_url	The test object request url.The parameters will be placed in string as {0}, {1} … Ex: www.tc.com/?gameId={0}&target={1}&hash={3}	String. Required.

A sample configuration file is also provided.

3.3 Dependencies Configuration

The Object Factory component needs to be configured with the keys needed to create the required instances. For details please see the Object Factory component specifications document.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Please see section below.

4.3 Demo

This demo will try to follow the use of the component, from creating the user interface of the extension to providing new event handlers and scripting objects.

4.3.1 Creating the tool band user interface

The first thing to do is to actually create the tool bar user interface and hook up various user control events to handlers.

A tool band should be created by extending the *ToolBand* class. The user interface can be created using the Visual studio designer. Several properties are available at design time like the title and size of the tool band.

The class must be marked with the *ExtensionAttribute* and the *Guid* attribute. The Extension attribute is needed to be recognized by the installer class and the *Guid* to be accessible as a COM object.

```
[Guid("F2E189BE-405E-45fb-98C2-026CFFEBF95B")]
[Extension(“MyToolBand”)]
public class MyToolBand: ToolBand
{
	MyToolBand()
	{
	}
}
```

The nest thing to do is get the client side logic for the Orpheus web application needed by the extension. This should happen right after the web browser reference is set. This is done in the *SetSite* method and the simplest way to do this is to override the method call *base.SetSite()* and then get the client logic object.

```
[Guid("F2E189BE-405E-45fb-98C2-026CFFEBF95B")]
[Extension(“MyToolBand”)]
public class MyToolBand: ToolBand
{
	//hold a reference to the client side logic object
	private MsieClientLogic clientLogic = null;

	MyToolBand()
	{
	}

	//override the method sa we know when the site is set.
	public override SetSite(pUnkSite object)
	{
		base.SetSite (pUnkSite);
```

```
            //after the site is set and we have a reference to the web browser
            we get the client side logic object

            this.clientLogic = MsieClientLogic.GetInstance(this.Host);

            //this should get the singleton instance of the object. The
            singleton is needed so the same MsieClientLogic instance is shared
            if a popup browser window is opened.

            The MsieClientLogic can also be created using the public
            constructors.
            this.clientLogic = new MsieClientLogic(this.Host);

            //or passing all the needed objects.

            //after the MsieClientLogic is created we need to customize the new
            browser. This class will be created for every new browser window,
            so there will be one instance of the MyToolBand class for every
            browser window, and, if using singleton, one instance of
            MsieClientLogic for all.

            //customize the opened browser window
            this.clientLogic.CustomizeWebBrowser(this.Host);

            //this will customize the browser with the same IDocHostUIHandler
            object and the same scripting object.
      }
}
```

The user interface controls should be added and hookup to event handlers.

```
private void btnLogIn_Click(object sender, System.EventArgs e)
{
      //create the event arguments class
      ExtensionEventArgs args = new ExtensionEventArgs(“LogIn”,
      this.clientLogic);

      //fire the corresponding event
      this.clientLogic.EventsManager.FireEvent(“LogIn”)

      //note that handlers (HttpRequestUserInterfaceEventHandler) should be
      configured to respond to this event.

}


//the event handlers should be configured in the configuration file
//event handlers can be configured for displayed page changed and updates
pooling events as well

//an event handler can also be set to a class method. When an event is fired by
the scripting object for example, we need to handle that.

//declare the handler method
public void HandleLoggedIn(object sender, ExtensionEventArgs args)
{
      //get the context
      MsieClientLogic context = args.Context;

      //handle event
```

```
}

//create a delegate for the method
ExtensionEventHandlerDelegate delegate =
new ExtensionEventHandlerDelegate(HandleLoggedIn);

//add the handler
this.clientLogic.EventsManager.AddEventHandler(“LoggedIn”, delegate);
```

4.3.2 Working with the scripting object

The component exposes to the web pages JavaScript an object through which JavaScript code can communicate with the component.

```
<script type="text/javascript">
       function ScriptingInterface()
{
       //signal logged in
       window.external.LoggedIn();

       //signal logged out
       window.external.LoggedOut();

       //set the working game id
       window.external.SetWorkingGame(<gameId>);

       //get working game
       Var gameid = window.external.GetWorkingGame();

       //set current target
       window.external.SetCurrentTarget(<hash>, <sequence>);

       //force get updates from server
       window.external.PollMessages();

       //check if window is popup
       var popup = window.external.IsPopup(window);

}
</script>
```

4.3.3 Working Programmatically with the Client Side Logic

```
//create a new client side logic object
MsieClientLogic logic = new MsieClientLogic(<web browser>);

//or as a singleton
MsieClientLogic logic = MsieClientLogic.GetInstance(<web browser>);


//or using custom configuration namespaces
MsieClientLogic logic = new MsieClientLogic(<web browser>, <custom config>,
<custom object factory>);

//get the browser
WebBrowserClass browser = logic.WebBrowser;

//get the events manager
IExtensionEventsManager events = logic.EventsManager;
```

```
//add a handler
events.AddEventHanlder(“event name”, <method delegate>);

//get all handlers for an event
ExtensionEventHandlerDelegate[] delegates=events.GetEventHandlers(“event name”);

//remove an event handler
events.RemoveEventHanlder(“event name”, <method delegate>);

//fire an event
events.FireEvent(“event name”);

//get the web browser window navigator
IWebBrowserWindowNavigator navigator = logic.WebBrowserWindowNavigator;
//navigate to a URL in a new window; false to navigate in the same browser
window
navigator.Navigate(logic.Host, <URL>, true);

//set the content to the web browser window from a stream
//get a stream to display in the browser
Stream doc = <assign stream>;
navigator.Navigate(logic.Host, doc, true);


//get the persistence
IPersistence storage = logic.Persistence;

//persist a value
storage[“key”] = “value”;

//get the value back
string value = storage[“key”];


//get or set the bloom filter
BloomFilter filter = logic.BloomFilter;

//get the browser customization object
IDocHostUIHandler  handler = logic.BrowserCustomization;

//get the scripting object
object scriptingObject = logic.ScriptingObject;
```

4.3.4 Installing the extension

In order for the *MyToolBand* class to be accessible to COM clients must it be registered with the Assembly Registration Tool.

regasm *assemblyFile*

To unregister:

regasm *assemblyFile* /u

The assembly and the dependent assemblies must be in either the same folder as Internet Explorer or added to the GAC in which case they must have a strong name.

4.3.5 *Implementing New Event Handlers*

All event handlers must implement the *IExtentionEventHandler* interface. Event handlers are added to the configuration file, and invoked by the component when the event they are configured for is fired.

```
public class MyEventHandler : IExtentionEventHandler
}
	MyEventHandler()

	public void HandleEvent(object sender, ExtensionEventArgs args)
	{
		//handle event
	}
}
```

The new event handler implementation must be configured for a specific event.

4.3.6 *Implementing New Scripting Objects*

Scripting objects have only to be marked with the *ComVisible* attribute and must provide a constructor that accepts an *MsieClientLogic* object to be created by the component. As a result the object properties and method will be available to JavaScript code.

To be created the new scripting object must be configured in the configuration file.

5. Future Enhancements

- Other implementations of *IExtensionEventsManager* can be plugged in.
- Other implementations of *IExtensionEventHandler* can be added as well as different *IExtensionEventHandlerFactory* implementations.
- Other *IWebBrowserWindowNavigator* and *IPersistence* implementations can be plugged in.
- Other *IDocHostUIHandler* and scripting objects can be implemented to customize the web browser, including providing context menu items.