PureConnect®

2018 R5

Generated:

12-November-2018

Content last updated:

27-April-2018

See Change Log for summary of changes.

GENESYS™

Interaction Desktop Add-in

Technical Reference

Abstract

This document provides a high-level overview of how and when to use the Interaction Desktop Add-In API to create add-ins. It includes guidelines for creating installers and deploying add-ins.

For the latest version of this document, see the PureConnect Documentation Library at: http://help.genesys.com/cic.

Table of Contents

Introduction to Interaction Desktop Add-In

This section covers:

- Scenarios in which you might want to write an add-in
- Conditions for writing an add-in
- Versioning an add-in
- Writing an installer for an add-in
- Deploying an add-in

Note:

While the client system is called Interaction Desktop, the namespace and the actual APIs are called InteractionClient i.e. `ININ.InteractionClient.AddIn Namespace`.

This technical reference provides a high-level view of scenarios and steps for creating add-ins. Detailed information about add-in classes, methods, and other technical features can be found in the API help, which is available on the PureConnect Developer Portal.

Add-In Scenarios

- Create a Screen Pop
- Create a Queue Monitor (respond to interaction adds/changes/removes)

Conditions

- Add-ins are not deployed or installed by PureConnect. Therefore, you must install and deploy them.
- If you create an add-in, you must create an install for it.
- You must provide your own error checking and security.
- The key difference between using IceLib and DDE (customization points) is that you use IceLib to create a standalone application. With an add-in, on the other hand, you must deploy your add-in DLL alongside the Interaction Desktop client.
- Screen pops are the only add-ins that require server-side configuration. Other add-ins, such as queue monitors, run only on the user's client.

Versioning

As with any public API, versioning is a concern. The Interaction Desktop Add-in API takes a conservative approach to versioning by using a specific API version number. This version is not the same as a file version number, which gets increased for each release. The API version number is increased when a breaking change is made to the public API. Note that if the API changes only in an additive, non-breaking way, then the version number will not increment. The version number will only increment when a breaking change is made.

> **Note:**
> A **breaking change** is defined as a change to the public API that would require an add-in to be recompiled to accommodate the change.

If they are required, breaking changes will only be made between full releases.

Interaction Desktop detects an add-in's version by the use of an assembly-level attribute, `ININ.InteractionClient.AddIn.AddInVersionAttribute`. The Interaction Desktop Add-in assembly, `ININ.InteractionClient.AddIn.dll`, contains this attribute specifying a specific version. Interaction Desktop will only load add-ins that match the current version specified in `ININ.InteractionClient.AddIn.dll`. The current version can be determined by using the `ININ.InteractionClient.AddIn.AddInVersion.CurrentVersion` field.

The version numbers must match to ensure that both the Interaction Desktop and the third-party add-in use the same API.

If an add-in exists with an incorrect version number, a trace message will be written into the Interaction Desktop's log file to help with diagnostics.

Add-In developers are strongly encouraged to use the `ININ.InteractionClient.AddIn.AddInVersion.CurrentVersion` field as described below in the "Creating Add-Ins" section. The field is defined as a const which means the value will be compiled into the Add-In at compile time, and when the API version changes, a recompilation of the Add-In will be required to pick up the new version. This allows developers to compile against an Add-In version without having to know the exact version number required.

Deploying an Add-in

Deploying a single add-in is relatively simple procedure. To do so, just copy your add-in's DLL to a subfolder called Addins under the directory in which Interaction Desktop is installed. For example,

```
C:\Program Files (x86)\Interactive Intelligence\ICUserApps\Addins
```

Or

```
C:\Program Files\Interactive Intelligence\ICUserApps\Addins
```

If the `\Addins` subfolder doesn't exist, create it. If the folder already exists, simply place your files in it.

> **Note:**
> If your add-in references other DLLs, they must also be in the folder.

Writing an Installer for an Add-in

You can use any of several tools or languages to create an install program for the add-ins that you develop. So, the only requirement is that your installer copies the DLL for your add-in, and any other DLLs that your add-in references, to the `\Addins` subfolder described above.

Creating Add-Ins

- First steps in creating an add-in
- Writing a screen pop add-in
- Writing a queue monitor add-in
- Writing an add-in for any purpose

First Steps to Create Any Add-In

To write any add-in, regardless of its type, first do this:

1. In Visual Studio, create a class library for the add-in.
2. In the Visual Studio project, add a reference to the file:
   `ININ.InteractionClient.Addin.dll`
   You can find the file in the Interaction Desktop installation directory.
3. In the file `AssemblyInfo.cs`, in the `Assemblies` section, add this line:
   `[assembly: AddInVersion(AddInVersion.CurrentVersion)]`

Writing a Screen Pop Add-In

To write a screen pop:

- In the main C# file (e.g., `Class1.cs`), find the main namespace, which will have the same name as the project itself.
- In the using section at the top of the C# file, add:

```
using ININ.InteractionClient.AddIn;
```

This makes the DLL's namespace available.

> **Tip:**
> When you type an identifier that occurs in the AddIn DLL, Visual Studio's Intellisense feature displays a list box of available properties, methods, or parameters for that identifier.

- Create a public class and give it a name related to the screen pop.

The name of the class does not matter. The new class should derive from `ININ.InteractionClient.AddIn.ScreenPop`, which is defined in the `ININ.InteractionClient.AddIn.dll` assembly.

- Override the base class's `Pop` method and `Name` property.

The original code (for a screen pop named "Foo") will look like this:

```
public class FooScreenPop : ScreenPop
{
  public override void Pop(IDictionary<string,string> attributes)
  {
    throw new NotImplementedException();
  }

  public override string Name
  {
    get { throw new NotImplementedException(); }
  }
}
```

The modified code (for a screen pop named "Foo") will look like this:

```
public class FooScreenPop : ScreenPop
{
  // Interaction Desktop will call this method according to its
  // corresponding screen pop configuration in Interaction
  // Administrator. The "attributes" parameter will contain (at
  // a minimum) all the name/value pairs that were specified in
  // Interaction Administrator.

  public override void Pop(IDictionary<string,string> attributes)
  {
    // For example:
    string customerId = attributes["customer-id"];

    // Then do some action with values from attributes.
    // For example:
    CallMySpecificApplication(customerID);
  }

  // override the inherited Name field
  // this must match the name in IA
  public override string Name
  {
    get { return "Foo"; }
  }
}
```

Writing a Queue Monitor Add-In

A queue monitor watches for specific interaction events in a queue and does actions based on those events. Writing a queue monitor is slightly more complex than writing a screen pop, but is still fairly straightforward.

The `ININ.InteractionClient.AddIn` namespace defines a `QueueMonitor` class with three methods for monitoring queues and performing actions in response to queue events:

- **InteractionAdded:** This method fires when an interaction is added to the queue. Inside this method, you can insert code to check attributes of the interaction and perform actions based on attribute values.
- **InteractionChanged:** This method fires when a queue interaction changes. Inside this method, you can insert code to check attributes of the interaction and perform actions based on attribute values.
- **InteractionRemoved:** This method fires when an interaction is removed from the queue.

To write a queue monitor add-in:

1. Create a class library as above. Derive it from the `QueueMonitor` abstract class in the `ININ.InteractionClient.AddIn` DLL.
2. In the main `.CS` file for the class, add this `USING` statement:
   ```
   using ININ.InteractionClient.AddIn;
   ```
   > **Tip:**
   > When you type an identifier that occurs in the AddIn DLL, Visual Studio's Intellisense feature displays a list box of available properties, methods, or parameters for that identifier.
3. Override the inherited QueueMonitor methods as shown in the code below:

```
public class MyInteractionMonitor :
 QueueMonitor
{
// Returns the list of attributes that
 you want to
// monitor: that is, the attributes
 for which you
// want to receive change notifications.
protected override IEnumerable<string>
 Attributes
{
get
{
// The return
 values here are *examples* of attributes.
return new
 string[]
{
InteractionAttributes.State,
"Eic_AnotherAttribute",
"Eic_SomethingElse",};
}
}
protected override void
 InteractionAdded(IInteraction interaction)
{
// code to
 check attribute value and perform action
}
protected override void
 InteractionChanged(IInteraction interaction)
{
string state
 = interaction.GetAttribute(InteractionAttributes.State);
// code to
 check attribute value and perform action
}
protected override void
 InteractionRemoved(IIinteraction interaction)
{
}
}
```

Examining the Queue Monitor Code

The `QueueMonitor`-derived class that you create watches interactions. In particular, it watches the attributes of interactions that you list in the Attributes property's return statement.

The attributes that you list in the `Attributes` property are the interaction's attributes that the add-in will monitor. The methods (`InteractionChanged`, etc.) will only be called in response to changes in those attributes.

Standard system attributes are in the `ININ.InteractionClient.AddIn.InteractionAttributes` static class. Using this class avoids the need to know exact attribute names. You can, however, specify any string value: for example, to watch and access a custom attribute.

If you don't remember the name of a standard attribute, Visual Studio's Intellisense feature will display a list of attributes from the helper class for you.

If you have listed an interaction attribute in the `Attributes` property, you can retrieve the value of that attribute for an interaction by using either the `GetAttribute` method or the indexer available on the `IInteraction` interface. For example:

```
string customerid = interaction.GetAttribute("Customer_ID");
```

or:

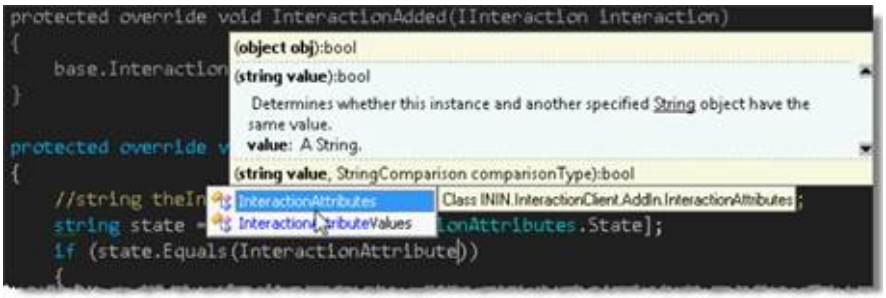
```
string customerid = interaction["Customer_ID"];
```

Here, you retrieve the value of the interaction's `Customer_ID` attribute and store the value in your add-in's `customerid` string variable.

Based on the value you retrieved, you could then make your queue monitor perform an action:

```
if (customerid.StartsWith("preferred-"))
{
// do something
}
```

When using standard system attributes such as `State`, you can use the `InteractionAttributes` static class when specifying the attribute to watch, and can use the `InteractionAttributeValues` static class to compare against known system values the attribute can contain.



Using this method, you might write:

```
string state = interaction.GetAttribute(InteractionAttributes.State);
if (state.Equals(InteractionAttributeValues.State.Connected))
{
// do something with the connected interaction
}
```

Writing a Custom Window Add-In

A custom window (or view) add-in for Interaction Desktop provides a client view that the user can select. It adds the custom view to Interaction Desktop's tab-based user interface.

Note that custom views implemented in an add-in do not automatically appear in the user interface. The user must access the **Create New View** dialog from the **File** menu and select the custom view. If users previously added the custom view, Interaction Desktop will automatically display it the next time they start their Interaction Desktop.

You can use a custom view in the Interaction Desktop to host any user control or custom control.

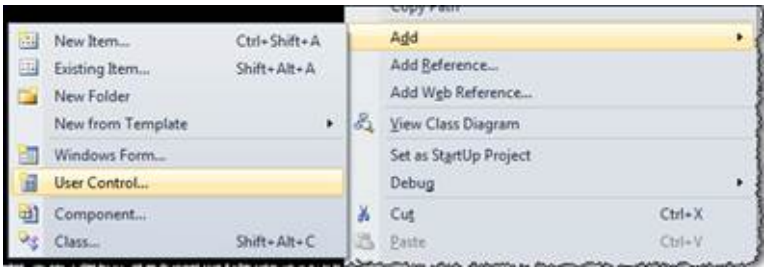
The `ININ.InteractionClient.AddIn` namespace defines an `AddInWindow` class with a number of abstract properties to override:

- **Id:** The unique identifier of this view. This is used, for example, when the Interaction Desktop persists the open *view* (tabs) during shutdown and re-creates each view on startup.
- **DisplayName:** The friendly name of the view. This is displayed in the **Create New View** dialog when the user is selecting which view to display in the Desktop client.
- **CategoryId:** The unique identifier of the view's category. If you are adding multiple custom views and want them to appear in the same category, this value must match for each view.
- **CategoryDisplayName:** The friendly name of the category. This is displayed in the **Create New View** dialog.
- **Content:** The user control or custom control to embed (docked to fill) inside the view when the view is created.

To write a custom view add-in:

1. In the main C# file (e.g., `Class1.cs`), find the main namespace, which has the same name as the project itself.
2. In the `using` section at the top of the C# file, add:
   `using ININ.InteractionClient.AddIn;`
   This makes the DLL's namespace available.

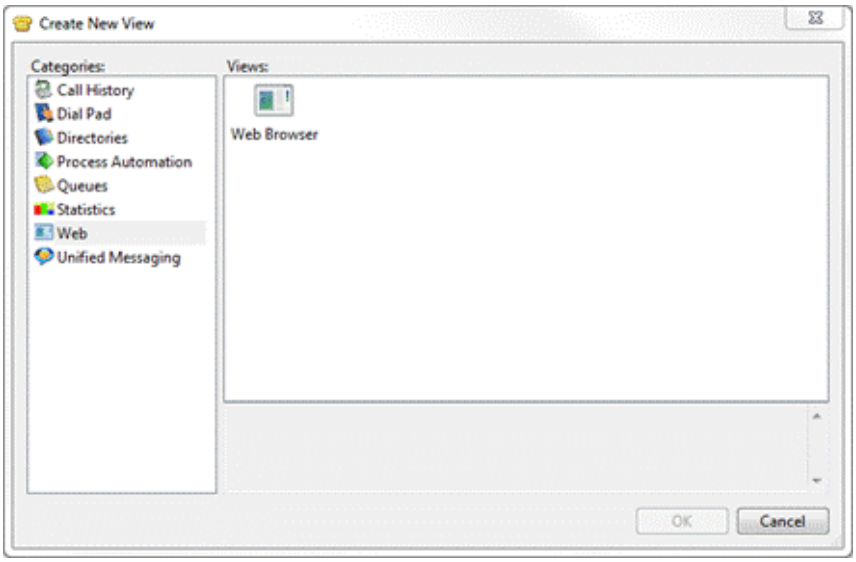
   Tip:
   When you type an identifier that occurs in the AddIn DLL, Visual Studio's Intellisense feature displays a list box of available properties, methods, or parameters for that identifier.
3. Create a public class and give it a name related to the custom window.
   The name of the class does not matter. The new class should derive from `ININ.InteractionClient.AddIn.AddInWindow`, which is defined in the `ININ.InteractionClient.AddIn.dll` assembly.
4. Right-click the project name (see figure), click **Add**, and click **User Control**.



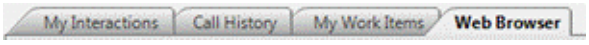
5. Finish creating the `UserControl` with the same method you normally use to create a `UserControl`. The example code in the next step of this procedure creates a `UserControl` that contains a `WebBrowser` control named `WebBrowserTabContent`.
6. Override the base class properties specified above.
   Add code for the `UserControl` that follows the pattern shown in this example:

```
public class WebBrowserAddInWindow : AddInWindow
{
  protected override string Id
  {
    get { return "MY_WINDOW"; }
  }

  protected override string DisplayName
  {
    get { return "Web Browser"; }
  }

  protected override string CategoryId
  {
    get { return "MY_CUSTOM_CATEGORY"; }
  }

  protected override string CategoryDisplayName
  {
    get { return "Web"; }
  }

  public override object Content
  {
    get { return new WebBrowserTabContent(); }
  }
}
```

   Creating an add-in as shown in the example results in a **Create New View** dialog.



After the user adds the **Web Browser** page, the new view is available in Interaction Desktop:



Writing an Add-In for Any Purpose

You can create an add-in for any purpose. Because add-ins provide specific, simplified support for creating screen pops and queue monitors, it's slightly more complicated to create an add-in for some other purpose. However, it's still fairly straightforward.

To write an add-in for some other purpose, you declare your add-in class and have it implement the `IAddIn` interface. The `IAddIn` interface has two methods:

- **Load:** When the Desktop client starts, it calls this method to load the add-in:

```
public void Load(IServiceProvider serviceProvider)
{
  // The code here runs when the client loads the add-in.
}
```

  The `serviceProvider` parameter enables the add-in code to retrieve services from the client.
- **Unload:** When the Desktop client shuts down, it calls this method to unload the add-in:

```
public void Unload()
{
  // The code here runs when the client unloads the add-in.
}
```

To write an add-in for a purpose other than a screen pop or queue monitor:

- In the main C# file (e.g., `Class1.cs`), find the main namespace, which will have the same name as the project itself.
- In the `using` section at the top of the C# file, add

```
using ININ.InteractionClient.AddIn;
```

This makes the DLL's namespace available.

> **Tip:**
> When you type an identifier that occurs in the AddIn DLL, Visual Studio's Intellisense feature displays a list box of available properties, methods, or parameters for that identifier.

- Create a public class with the name of the add-in and make it implement the `IAddIn` interface.

For example, to display a popup notification window when the add-in loads, you might write:

```
public class MyCustomAddin : IAddIn
{
  public void Load(IServiceProvider serviceProvider)
  {
    INotificationService notification = (INotificationService)serviceProvider.GetService(typeof(INotificationService));
    notification.Notify("My add-in loaded!", "Success!", NotificationType.Info, TimeSpan.FromSeconds(10));
  }
}

public void Unload()
{
  // The code here runs when the client unloads the add-in.
}
```

The pattern for using any Add-In service is the same as shown above for the notification service. You can find available services in the `ININ.InteractionClient.AddIn` namespace. You can find more information in the API documentation.

Finishing Up

After you have written an add-in of any type, build the project in Visual Studio just as you normally would. Then install the add-in and make it available to the Desktop client.

To Learn More

To learn more about specific features, explore *Programmable Add-In Help* and XML files:

- `ININ.InteractionClient.AddIn.chm`
- `ININ.InteractionClient.AddIn.xml`

You can find those files in the Interaction Desktop installation directory.

Appendix A: Creating Custom Secure Input Forms

You can use the Add-In API to implement custom secure input forms. For more information, see *Secure Input Technical Reference*.

Code listing A-1 gives an example of the API. It consists of the interfaces `ISecureInput` and `ISecureInputForm`.

To create a custom secure input form:

- Write code that implements the `ISecureInput` interface.
- Register that implementation with the `ISecureInputService`

Interaction Desktop uses the `IsecureInputService` to find and display the selected form when the agent clicks the **Secure Input** toolbar button.

Listing A-1: Example of API for custom secure input forms

```
namespace ININ.InteractionClient.AddIn
{
  /// <summary>
  /// Provides access to a custom secure input. Implement this
  /// interface and add an instance of the implementation
  /// to the <see cref="ISecureInputService"/> to make the custom
  /// secure input available.
  /// </summary>
  /// <remarks>
  /// A secure input is selected by the user. If the user selects a custom secure input,
  /// the Interaction Desktop will use the matching named form which was added to the
  /// <see cref="ISecureInputService"/>.
  /// Parameters configured by an administrator dictates what information is passed
  /// into the <see cref="GetForm"/> method. In addition, interaction attributes
  /// (if available) specified by the <see cref="AdditionalAttributes"/> property will be
  /// included in the parameter dictionary given to the <see cref="GetForm"/> method.
  /// </remarks>
  ///
  public interface ISecureInput
  {
  /// <summary>
  /// Gets the name of this secure input. The name is used to find the secure input and activate it.
  /// </summary>
  /// <value>The name of the secure input.</value>
  string Name { get; }

  /// <summary>
  /// Gets additional interaction attributes this secure input needs. These attributes
  /// will be gathered and included in the parameter passed to the <see cref="GetForm"/> method
  /// in addition to server-defined parameters.
  /// </summary>
  /// <value>The interaction attributes to retrieve.</value>
  IEnumerable<string> AdditionalAttributes { get; }

  /// <summary>
  /// Gets the secure input form to display.
  /// </summary>
  /// <param name="parameters">The specified Interaction and secure input
  /// parameters.</param>
  ISecureInputForm GetForm(IDictionary<string, string> parameters);
  }
}
namespace ININ.InteractionClient.AddIn
{
  /// <summary>
  /// A secure input form is a control (Windows Forms or WPF) that is embedded
  /// into a containing window and shown to the user in order to collect information
  /// before sending a caller into a secure IVR session.
  /// </summary>
  public interface ISecureInputForm
  {
    /// <summary>
```

```
    /// Gets a value indicating the state of the form.
    /// </summary>
    /// <returns><see langword="true"/> if the data is valid and the user may proceed,
    /// <see langword="false"/> if the user is not allowed to proceed with the
    /// secure input form.</returns>
    bool IsValid { get; }

    /// <summary>
    /// Occurs when the <see cref="IsValid"/> property is changed.
    /// </summary>
    event EventHandler IsValidChanged;

    /// <summary>
    /// Gets a dictionary with the name/value pairs that will be provided to the secure IVR,
    /// and to the 3rd party service processing the secure input session.
    /// </summary>
    IDictionary<string, string> SecureParameters { get; }

    /// <summary>
    /// Gets the main content to be displayed in the secure input window displayed to the user.
    /// </summary>
    /// <returns>
    /// Return either a Windows Forms Control or a Windows Presentation Foundation Control.
    /// Any other return type will be ignored.
    /// </returns>
    object Content { get; }
  }
}
```

With the API in Listing A-1, you can define three classes needed for secure input:

- `SecureInputAddin.cs`: Listing A-2. This class defines the Interaction Desktop add-in and implements the `IAddIn` interface.
- `CustomSecureInput.cs`: Listing A-3. This class implements the `ISecureInput` interface.
- `MyForm.cs`: Listing A-4. This (partially implemented) class provides content for the custom secure input form.

Listing A-2: The SecureInputAddin Class

```
namespace CustomSecureInputForm
{
  public class SecureInputAddin : IAddIn
  {
    public void Load(IServiceProvider serviceProvider)
    {
      var secureInputService = serviceProvider.GetService(typeof(ISecureInputService)) as ISecureInputService;
      if (secureInputService == null) return;
      secureInputService.Add(new CustomSecureInput());
    }
    public void Unload()
    {
    }
  }
}
```

This class defines the Interaction Desktop add-in and implements the `IAddIn` interface.

When Interaction Desktop loads the add-in, Interaction Desktop's Plug-in architecture calls the Load method. The `CustomSecureInputForm` class uses the `IServiceProvider` that the `Load` method provides in order to:

- Retrieve the `ISecureInputService`
- Register a custom implementation of the `ISecureInput` interface with the Interaction Desktop's `ISecureInputService`. In this case, the custom implementation is an instance of the `CustomSecureInput` class.

Interaction Desktop later uses `IsecureInputService` to access any of these custom secure input form implementations.

Listing A-3: The CustomSecureInput Class

```
namespace CustomSecureInputForm
{
  public class CustomSecureInput : ISecureInput
  {
    public string Name
    {
      get
      {
      return "CustomCreditCardProcessing";
      }
    }

    public IEnumerable<string> AdditionalAttributes
    {
      get { return new string[0]; }
    }

    public ISecureInputForm GetForm(IDictionary<string, string> parameters)
    {
      var formProvider = new MyForm { ConnectionString =
        parameters["database_connection_string"] };
      return formProvider;
    }
  }
}
```

This class implements the `ISecureInput` interface.

The `Name` property defines the name of the secure input form and must match the form name used in Interaction Administrator. Use the `AdditionalAttributes` property to specify any other interaction attributes required by the custom secure input form.

In the final section of code, Interaction Desktop calls the `GetForm` method when the agent clicks the **Secure Input** button and selects a secure input form. Interaction Desktop gets an `IDictionary` of key-value pairs corresponding to the custom parameters defined for the custom secure input form in Interaction Administrator. The example defines a database connection string in Interaction Administrator as one of a form's `Custom` parameters. The parameter is named `data_base_connection_string`.

Listing A-4: The MyForm Class

```
namespace CustomSecureInputForm
{
  public partial class MyForm : UserControl, ISecureInputForm
  {
    private bool _isValid;

    public MyForm()
    {
      InitializeComponent();
      IsValid = false;
      SecureParameters = new Dictionary<string,  string>(StringComparer.OrdinalIgnoreCase);
    }

    public object Content
    {
      get
      {
        return this;
      }
    }

    public bool IsValid
    {
      get
      {
        return _isValid;
      }
```

```
      private set
      {
        if (_isValid == value) return;

        _isValid = value;

        var evt = IsValidChanged;
        if (evt != null)
        {
          evt(this, EventArgs.Empty);
        }
      }
    }

    public event EventHandler IsValidChanged;

    public string ConnectionString { get; set; }

    public IDictionary<string, string> SecureParameters { get; private set; }

    private void btnLoad_Click(object sender, EventArgs e)
    {
      // Hit a web service here, or something, to retrieve the values for the specified customer ID
      string amount = "$ 142.12";
      string address = String.Format("7601 Interactive Way{0}Indianapolis, IN{0}46278", Environment.NewLine);

      lblAmount.Text = amount;
      lblAddress.Text = address;

      SecureParameters["amount"] = amount;
      SecureParameters["address"] = address;

      IsValid = true;
    }
  }
}
```

This partial implementation of the `ISecureInputForm` interface uses a WinForms `UserControl` that provides the content for a custom secure input form.

The code defines a mock form that pretends to access a database or web service to retrieve values with which to populate two form fields. After the **Load** button is pressed:

- Data loads into two labels on the form
- The `SecureParameters` dictionary is updated with the new values
- The `IsValid` property is set to `true`.

Those actions inform Interaction Desktop that prerequisite user input or activity is complete and that it can start the Secure IVR.

Change Log

This change log covers the initial release version of this document.

Date	Changes
16-November-2010	In the section "To Learn More," added the location of API help and XML files on the support site.
01-December-2010	Added section on writing a custom window.
30-March-2011	Updated the section "To Learn More" with the new location of the help files.
03-November-2011	Updated for IC 4.0.
15-February-2012	Updated two figures in the section "Writing a Custom Window Add-In."
10-August-2012	Added appendix about creating secure input forms.
15-August-2014	Updated documentation to reflect changes required in the transition from version 4.0 SU# to CIC 2015 R1, such as updates to product version numbers, system requirements, installation procedures, references to Interactive Intelligence Product Information site URLs, and copyright and trademark information.
22-June-2015	• Updated cover page to reflect new color scheme and logo. • Updated copyright and trademark information.
08-October-2015	• Changed the focus and name of the document from Interaction Client Programmable Add-in to Interaction Desktop Add-in. • While the client system is called Interaction Desktop, the namespace and the actual APIs are called `InteractionClient` i.e. `ININ.InteractionClient.AddIn` Namespace.
04-February-2016	Updated documentation to reflect 2016 R2 Release
27-April-2018	Rebranded from Interactive Intelligence to Genesys.