SIEBEL

Siebel RTD

Integration with Siebel RTD

Version 2.1 Document Revised September, 2005

Integration with Siebel RTD

Section 1: Overview 2

1.1 Choosing the best means of integration 2

1.1.1 About the Java Smart Client 2

1.1.2 .NET Smart Client 3

1.1.3 Web Services 3

1.2 About the Cross Sell Inline Service 3

1.2.1 Using Studio to Identify Object IDs 3

1.2.2 Determining the Response of an Advisor 4

1.2.3 Knowing how to respond to the server 5

1.2.4 Identifying Session Keys and Arguments 5

1.3 Using the Java Smart Client 6

1.4 Before you get started 6

1.5 Integrating with an Inline Service using the Java Smart Client 6

1.5.1 Java Smart Client API Reference 6

1.5.2 Preparing the Smart Client example 6

1.5.3 About Smart Client properties 8

1.5.4 Creating the properties file 8

1.5.5 Creating the Smart Client 9

1.5.6 Creating the Request 12

1.5.7 Invoking the Request 13

1.5.8 Examining the response 14

1.5.9 Closing the loop 15

1.5.10 Closing the Client 15

Section 2: Using Smart Client JSP tags 16

2.1 Integrating with an Inline Service using the JSP Smart Client tags 16

2.1.1 JSP Smart Client tag Reference 16

2.1.2 Preparing the Smart Client example 16

2.1.3 Sample JSP code 16

Section 3: .NET Smart Client Integration 17

3.1 Using the .NET Smart Client 17

3.1.1 .NET API Reference 17

3.1.2 .NET Integration Example 17

Section 4: Zero Client Integration 18

4.1 About Web Services 18

Preface

Purpose of this Guide

This guide provides information about integrating with Siebel RTD. It shows several methods of integration including: Siebel RTD Smart Clients, Web Services and directly messaging the Siebel Real-Time Decision Server.

Intended Audience

This document is intended for developers who will use the Java based API, .NET component or Web services to integrate deployed Inline Services running on Siebel RTD with enterprise applications. Integrating inline services does not require the full mastery of programming concepts with Java, .NET or Web services that software developers need, but rather an understanding of how those protocols can be used for application integration purposes. Integrators should be familiar with using Siebel Decision Studio and Inline Services as well as programming with Java, .NET or using Web Services.

Information about using Siebel Decision Studio to deploy Inline Services can be found in *Getting Started with Siebel RTD.*

How to use this guide

This document is divided into the following sections: **Section 1: Overview** provides an overview of the methods for integration toSiebel RTD. **Section 2: Using the Java Smart Client** gives a step-by-step walkthrough for using the Java Smart Client to integrate to an Inline Service. **Section 3: Java Smart Client API** explains the full API of the Smart Client. **Section 4: .NET Smart Client Integration** gives a step-by-step walkthrough for using the .NET Smart Client to integrate to an Inline Service. **Section 5: .NET Smart Client API** explains the full API of the Client. **Section 6: Zero Client Integration** explains the SOAP interface for Siebel RTD Decision Service.

Document conventions

Convention	Description
`monospace`	Indicates source code and program output.
bold	Indicates portions of the user interface, including labels, tabs, menus, etc.
italic	Italics are used to highlight the first use of terms.
‘quote’	Indicates input required from the user.
💡	Indicates additional information that may make the task easier.
📝	Indicates additional information about the subject.
⚠	Indicates actions that may result in loss of data or errors.

Section 1: Overview

Siebel RTD features various robust and easy to use means to integrate with enterprise operational systems:

- Smart Clients: For Java and .NET environments these components manage communication to Integration Points on the Siebel Real-Time Decision Server.
- Zero Clients: Access to Integration Points is available via Web Services as a zero client approach.

This document outlines how to use these means to integrate with deployed Inline Services running on Siebel RTD.

1.1 Choosing the best means of integration

Siebel RTD offers multiple means of integration. To choose the best means for your environment you should consider the platform you are working on, performance needs and the additional functionality offered by Siebel RTD Smart Client over other methods of integration.

1.1.1 About the Java Smart Client

The Siebel RTD Smart Client for Java is a component that allows easy, managed integration to deployed Inline Services for operational systems. If you are working in a Java environment, the Java Smart Client is the preferred means of integration. The Java Smart Client offers two important features above and beyond the other methods of integration: session affinity and default response values.

The factory methods of the Smart Client interface take parameters representing the minimal information required to establish contact with a cluster of servers. After connecting, the component's full configuration is downloaded from the server. This way only a small set of parameters must be managed in the client application, while most of the component's configuration is centrally managed by the server's administration console.

The configuration information returned by the server to the client is shared by all the instances of the Smart Client created in the same Java virtual machine. There is a client-side class called a client-side dispatcher that manages this shared configuration and also manages session-affinity information used to dispatch requests to the correct server, based on session keys in the request.

The Smart Client is thread-safe, but for optimal performance a separate Smart Client should be created for each thread. Separate instances of the Smart Client share information and connections, so there is practically no penalty to having multiple instances.

The Smart Client is thread-safe, but it serializes execution of requests to the server. To achieve concurrent execution of requests, a separate SDClientInterface object should be created for each thread.

Several factory methods are available to create a Smart Client. Most either directly or indirectly reference a properties file in the file system or on a web server. The properties file supplies addresses for connecting to one or more servers in a single cluster as well as other properties that configure the connection to the server. Factory methods are also available to directly supply a HTTP URL and port or use a default address.

After the client's constructor communicates with one server and receives more complete configuration information, the detailed configuration is saved in a local file called the *client configuration cache*, where it can be accessed should the client restart when the server is unavailable. The configuration cache contains information such as the current list of hosts deployed to the server's cluster, and the client's set of default responses. The client's configuration cache is updated automatically by the client whenever it changes in the server.

Part of the configuration information downloaded to a client from the server includes a set of default responses to use if the client loses contact with the server or the server fails to respond to an integration point request in a timely fashion. This maintains the Service Level Agreement (SLA) between the Siebel Real-Time Decision Server and client application regardless of individual transactional availability.

These default responses are configured at the granularity of the individual integration points; each integration point relies on its own specialized default response. When any default responses are reconfigured on the server, the changes are propagated automatically to the client's out-of-band data, bundled together with normal integration point responses.

The Java Smart Client automatically maintains session affinity and routes requests to the correct server in a cluster. It also automatically and dynamically reconfigures itself with default responses to be used in case the server becomes overloaded or otherwise unable to abide by its configured level of service.

To achieve clustering using the other methods of integration you can use a third party clustering solution.

1.1.2 .NET Smart Client

For the .NET environment a .NET Smart Client component is available. This component offers a way to call the same interfaces provided by the Java Smart Client. However, it does not offer the added functionality of maintained session affinity or default values. This functionality will be available in the near future.

1.1.3 Web Services

Any client can access the Siebel Real-Time Decision Server through Web Services. The benefit to this means of integration is the lack of code needed on the client. Web service operations are defined in a WSDL file and definitions are contained in a schema file.

1.2 About the Cross Sell Inline Service

Example Inline Services are included with Siebel Decision Studio. One of these is a cross selling example.

The Cross Sell Inline Service simulates a simple implementation for a credit card contact center. As calls come into the center, information about the customer and the channel of the contact is captured.

Based on what we know of this customer, a cross selling offer is selected that is extended to the customer. The success or failure of that offer is tracked and sent back to the server so that the underlying decision model has the feedback that helps to refine its ability to make a better cross selling recommendation.

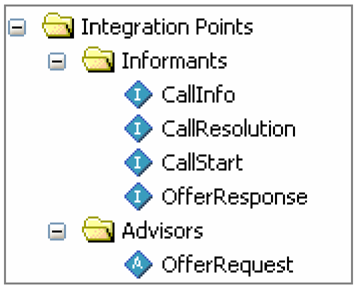
The Cross Sell Inline Service is used to demonstrate the various means of integration in this guide.

Included in the Cross Sell example are several integration points. Use the following instructions to familiarize yourself with these.

Informants execute on the server when supplied with the proper parameters. Advisors execute and also return data. In order to supply the correct parameters for calls to Integration Points, we must first identify the *Object IDs*.

1.2.1 Using Studio to Identify Object IDs

1. Using the **Start** menu, select **Studio** from the **Siebel Analytics→RTD** program group.
2. Select **File→ Import** to open the CrossSell Inline Service. **Import** appears.
3. Select **Existing Project into Workspace** and click Next. Browse for the Cross Sell project at the location $INSTALLDIR\examples\CrossSell. Select **OK** and click **Finish**, opening the project.
4. Using the Inline Service Explorer, expand **Integration Points. Informants** and **Advisors** are listed below. Expand each of these to view the Integration Points. Use the Object ID toggle to show the Object ID in the Inline Service Explorer. When the toggle is highlighted, the Object IDs show in the Explorer; when not, the display label is shown.



The Object ID of the Integration Point may or may not be the same as the label. Object IDs are used to reference Integration Points in method calls.

1.2.2 Determining the Response of an Advisor

Integration Points that deliver responses are called Advisors. An Advisor's **Response** tab in Studio determines the response, by identifying a parameterized Decision object that gets implicitly invoked by the Advisor. The Decision object's responsibility is to select the best Choices from its assigned Choice Group. The choice attributes that are returned are determined by the configuration set on the definition of the Choice Group.

In our example the OfferRequest integration point is an Advisor. It returns a single cross sell offer when it is invoked.

1. In Studio, select the OfferRequest Integration Point to view the editor.
2. On the **Response** tab, under **Decision,** look up the Decision that OfferRequest uses to return a response. It should be **OfferDecision**.
3. Double click OfferDecision under **Decisions** to view its detail pane.
4. On the **Selection Criteria** tab, under **Number of Choices to Select** find the number of responses that OfferRequest provides.
5. On the **Selection Criteria** tab, under **Choice Group,** find the Choice Group that OfferRequest uses. It should be Offers.

6. Under **Choices** double click **Offers** to see the choice attributes associated with this Choice Group. These attributes will be returned when a call to the Advisor is made.

> TIP In Studio, use the **Test** view to call the advisor and see what is returned. This way you will see the offer returned and the attributes that come with it. **Test** is available using the tab beside **Problems**. Use the **Execute Request** button to send the request to the server.

1.2.3 Knowing how to respond to the server

Inline Services are most powerful when the success or failure of a Choice is tracked and the model is self learning based on that information. In order to know what feedback the server needs to be self learning you must examine the Choice Event Model.

1. In Studio, double-click the **Offer Acceptance** Choice Event Model. The editor will appear on the right.
2. On the **Choice** tab, under **Positive Outcome Events** you see the Events that the server is interested in for learning. These are:
   - Interested
   - Purchased

   These outcomes are to be reported to the server from your Inline Service to give the proper feedback to the model.
3. The **OfferResponse** integration point is responsible for reporting this information.

1.2.4 Identifying Session Keys and Arguments

To invoke an Integration Point, we must supply values for the session keys and arguments expected by the Integration Point. In the request, we must use the Object IDs defined by Studio for the Integration Point's session keys and arguments. The key name must match one of the session key names defined in Studio for the Integration Point.

1. Select the **CallStart** Integration Point. On the **Request** tab of the editor of the integration point, under the **Session Keys** list, a path to the session key is shown starting with 'session'; the last name in the path is the Object ID of the session key.

> Note: If the session key is not displayed in object format use the **Object ID toggle** to change the display settings. Only the final object ID is necessary for the session key. For example, in the case shown above only the final string, `customerId`, is used.

2. To identify the arguments of the Integration Point, use the detail pane of to view the **Incoming Attribute** column of the **Request** tab. The **CallStart** incoming argument is **channel**.

1.3 Using the Java Smart Client

This section introduces using the Java Smart Client for integration. An example is included with Siebel RTD installation.

1.4 Before you get started

In order to work with the following example, you should have the following:

1. An installed Java Development Kit (JDK), with the JAVA_HOME environment variable set to its location. To obtain a JDK, go to the Sun Microsystems website, http://java.sun.com/products/.

2. The Cross Sell Inline Service deployed. For more information on deploying an Inline Service, see the *Siebel Decision Studio Reference Guide.*

3. The Siebel Real-Time Decision Server is started. For more information on starting the Siebel Real-Time Decision Server, see *Getting Started with Siebel RTD.*

1.5 Integrating with an Inline Service using the Java Smart Client

In general, integration using the Smart Client includes the following steps:

1. Prepare a properties file.

2. Create a connection to the Inline Service.

3. Create a request that identifies the integration point to connect to and the parameters to identify the session and any other information the integration point needs to determine an outcome.

4. Invoke the request.

5. Gather and parse any response information from Advisors.

6. Close the connection.

1.5.1 Java Smart Client API Reference

A reference to the Java Smart Client API for integration is available through the Siebel Decision Studio online help system.

1.5.2 Preparing the Smart Client example

For this example, the Cross Sell Inline Service has been integrated to a simple command line application to demonstrate how to use the Smart Client for integration.

Use the following steps to prepare the Smart Client example.

1 Edit the file $INSTALLDIR\client\sdbootstrap.properties

```
AppServer=jboss

rootDir=c:/temp

log.file=c:/temp/sdclient.log

sdtarget=all
```

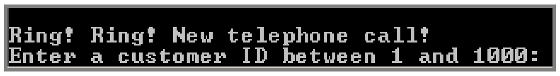
```
client=true
```

Ensure that `rootDir` and `log.file` are pointing to existing temporary directories. Save the file.

2 From within Studio, use **File→Import** menu to import the project from $INSTALLDIR\client\Client Examples\Java Client Example.

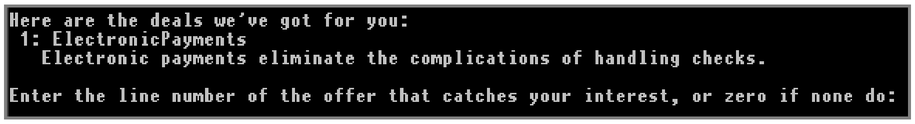
3 Use **Run→Run** to set up the Run environment. Choose the Java Application configuration. Browse for the main class (Example.java) and name the configuration. Click **Run.** The application does the following:

- Simulates a phone call.
- Creates a Smart Client for the Cross Sell Inline Service.
- Takes a customer ID from the command prompt.



```
Ring! Ring! New telephone call!
Enter a customer ID between 1 and 1000:
```

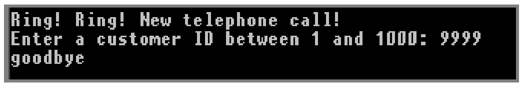
- Creates two requests (for CallStart and CallInfo) and populates them with the session key (customer ID) and a channel for the contact.
- Invokes the requests, thereby sending the information to the server.
- Creates another request (for OfferRequest) to get an offer.
- Invokes the request.
- Parses the response from the server.
- Presents the offer to the customer.



```
Here are the deals we've got for you:
 1: ElectronicPayments
   Electronic payments eliminate the complications of handling checks.

Enter the line number of the offer that catches your interest, or zero if none do:
```

- Captures the customer's interest in the offer made.
- Creates another request and populates it with the customer's interest (1) or non-interest (0).
- Invokes the request, sending the learning information back to the server.



```
Ring! Ring! New telephone call!
Enter a customer ID between 1 and 1000: 9999
goodbye
```

- Closes the Smart Client.

The source code for this example is found in the file $INSTALLDIR\client\examples\src\standAloneSmartClientCrossSell\Example.java, and is explained below.

1.5.3 About Smart Client properties

When a client application creates a Smart Client, it passes a set of properties to a Smart Client factory that represents the component's endpoint configuration. This file contains just enough information to allow the client to connect to a server endpoint. There are additional factory methods that use default configuration values; however it is best to explicitly specify the properties. The default properties file is shown below.

The factory method uses the properties to connect to the server. When the factory connects to the server, it downloads the more complete configuration information to the client, such as the set of default responses that the client should use if it ever needs to run when the server is unavailable. The detailed client configuration is saved in a local file, the Smart Client configuration cache, and is updated automatically whenever the server's configuration changes.

1.5.4 Creating the properties file

1. Open the properties file located at $INSTALLDIR\client\sdclient.properties.
2. Modify the contents to match your server configuration. Explanations of the elements of this file are listed below. In particular, make sure that you have a valid cache directory and the endpoint URL is the URL and port of your local Siebel Real-Time Decision Server. By default this is http://localhost:8080.

```
UseEndpointsInOrder = HTTP1

appsCacheDirectory = file:${rootDir}/temp

outOfBandRequestSeconds = 5

timeout = 0

HTTP1.type = http

HTTP1.url = http://localhost:8080/
```

Element	Description
`UseEndpointsInOrder`	A comma-separated list of endpoint names, indicating the order in which the endpoints should be tried when establishing an initial connection to the server cluster during the Smart Client's initialization. After initialization, this list of endpoints is irrelevant because the server will supply an updated list of endpoints. The endpoint names in this list refer to definitions within this properties file; the names are not used elsewhere.
`appsCacheDirectory`	A file URL identifying a writable directory into which the client component may save the configuration information that it gets from the server. The cache provides insurance against the possibility that the Siebel Real-Time Decision Server might be unavailable to the client application when the application initializes its client components. If sdclient.properties specifies a cache directory, it must already exist, otherwise, the client will use the Java virtual machine's temp directory
`outOfBandRequestSeconds`	The minimum interval, in seconds, between updates to the client's session affinity routing information. The routing information updated by this out-of-band data is shared by all Smart Client components created in the same Java virtual machine, so this interval governs how often the VM is updated, not each client separately. The out-of-band information also includes changes to the default responses configured for any inline services deployed to the server.
`timeout`	The timeout, in milliseconds, used by the original attempt to contact the server during the client component's initialization. After connecting to the server, the client uses the server's timeout, configured through JMX property, EntryPointRequestTimeout .
`<endpointName>.type`	The named endpoint type. Only HTTP is supported at this time.
`<endpointName>.url`	A URL specifying the HTTP host and port of the server's HTTP endpoint. The default endpoint is http://localhost:8080.

1.5.5 Creating the Smart Client

1. Open the source file for the Example application at `$INSTALLDIR\client\examples\src\standAloneSmartClientCrossSell\Example.java`

TIP This example source code can be used as a template for your Smart Client implementation.

2 The following imports are used to support Siebel RTD integration:

```
import com.sigmadynamics.client.IntegrationPointRequestInterface;
import com.sigmadynamics.client.IntegrationPointResponseInterface;
import com.sigmadynamics.client.ResponseItemInterface;
import com.sigmadynamics.client.SDClientException;
import com.sigmadynamics.client.SDClientFactory;
import com.sigmadynamics.client.SDClientInterface;
```

3 In the main method, the Example application demonstrates several techniques for using SDClientFactory to create an implementation of `SDClientInterface` based on the arguments supplied to the Example application.

These arguments are passed to `getClient`, where the proper factory method is identified.

```
SDClientInterface client = getClient(args);
```

There are several factory methods used to create a Smart Client. By examining `getClient`, we see the various methods:

```
private static SDClientInterface getClient(String[] args ){
    try{
      if ( args.length == 0 )
        return getClientWithDefaultPropertiesFile();
```

Creates a Smart Client with the default properties file using create(java.lang.String) The default properties file is referenced above.

```
      if ( "-h".equals(args[0])){
        if ( args.length < 2 )
          return getClientWithDefaultHttpAddress();
```

Creates a Smart Client with the default HTTP address of http://localhost:8080. This is the default installation url and port of the Siebel Real-Time Decision Server.
Uses createHttp(java.lang.String, int, boolea

```
        return getClientWithHttpAddress( args[1]);
      }
```

Creates a Smart Client with a supplied HTTP address. This is the address and port of your Siebel Real-Time Decision Server if it is not at the default address.
Uses createHttp

```
      if ( "-u".equals(args[0])){
        if ( args.length < 2 )
        {
          System.out.println("Missing properties file URL
          argument" );
          System.exit(-1);
        }
        return getClientWithPropertiesFileURL( args[1] );
      }
```

Creates a Smart Client with the information supplied in the properties file at the address specified.
Uses _create_1

```
      if ( "-f".equals(args[0])){
        if ( args.length < 2 )
        {
          System.out.println("Missing properties filename
argument" );
          System.exit(-1);
        }
        return getClientWithPropertiesFileName( args[1] );
      }
```

Creates a Smart Client with the information supplied in the properties file.
Uses _createFromPropertiesURL

```
      System.out.println("Unrecognized argument");
    }catch (SDClientException e ){
      e.printStackTrace();
    }
    System.exit(-1);
    return null;
  }
```

These methods are summarized in the section _Java_Smart_Client_API.

1.5.6 Creating the Request

1 The client application next creates a request to send to the Siebel Real-Time Decision Server.

`SDClientInterface` is used to create a request object:

```
createRequest(String appName, String integrationPointName);
```

> The appName parameter is the name of a server-resident application, developed in Studio.
>
> The integrationPointName parameter is the name of the application's informant or advisor that is to receive the request.
>
> See Using Studio to Identify Object IDs above to locate these values.

In our example, the request is created here:

```
IntegrationPointRequestInterface request  =
client.createRequest(INLINE_SERVICE_NAME, "CallStart");
```

2 The request object provides a method to set a single session key; call it separately for each key.

```
void setSessionKey(String keyName, String keyValue);
```

> By example, if an integration point's session key is listed in Studio as **session.customer.customerId**, you would pass **customerId** as the key name to `setSessionKey`.
>
> See Identifying session keys and argume above to locate these values.

In the example application the request session key is populated using:

```
request.setSessionKey( SESSION_KEY, sCustID );
```

where SESSION_KEY was set

```
static final String SESSION_KEY = "customerId";
```

and `sCustID` was captured from the command line input.

3 The request object provides two methods to set a single argument; call the appropriate one separately for each argument. The first method accepts an argument having a string value. The second method accepts an array of string values.

```
void setArg(String argName, String argValue);

void setArg(String argName, String[] argValue);
```

> The argument name should match one of the input names listed in Studio for the integration point.
>
> See Identifying session keys and argume above to locate these values.
> The value of this argument should be determined from the application design.

In the example application the request is populated using:

```
request.setArg( "channel", "Call");
```

1.5.7 Invoking the Request

1. After populating the request, the client application calls the `invoke` method of `SDClientInterface` to send the request to the server and receives an `IntegrationPointResponseInterface` representing an array of choices calculated by the server.

   ```
   IntegrationPointResponseInterface
   invoke(IntegrationPointRequestInterface request);
   ```

   In the example application this call is made:

   ```
   client.invoke(request);
   ```

   Note: If the client application wants to send a request for which it doesn't expect a response, and for which message delivery sequence is not critical, it can use the `invokeAsync` method instead of `invoke`.

   invokeAsync

   Requests sent via `invokeAsync` are not guaranteed to arrive at the server before requests sent via subsequent `invokeAsync` or `invoke` calls. When message delivery sequence is important, the `invoke` method should be used instead of `invokeAsync`, even when no response is expected.

2. After the request to the CallStart integration point is invoked a new request is prepared and invoked for CallInfo.

   ```
   // Supply some additional information about the telephone call.

   // Apparently the CrossSell service expects very little here --

   // just the channel again, which it already knows. Hence this message
   // could be left out with no consequences.
   request = client.createRequest(INLINE_SERVICE_NAME, "CallInfo");
   request.setSessionKey( SESSION_KEY, sCustID );
   request.setArg( "channel", "Call");
   client.invoke(request);
   ```

1.5.8 Examining the response

When an Advisor is invoked, a number response items, also known as Choices, will be returned. Your application must be prepared to handle this number of response items. See _Determining_the_response_of an Advi above.

In the client application, the selected Choices are accessible through the `IntegrationPointResponseInterface` returned by the invoke method. The `IntegrationPointResponseInterface` provides access to an array of response item objects, `ResponseItemInterface`, where each response item corresponds to a Choice object selected by the Advisor's Decision.

_com.sigmadynamics.client__Interface surfaces a Choice as a collection of value strings, keyed by name string.

In our example, the response is examined as such:

1. When invoking a request on an Advisor integration point, be prepared to receive a response.

```
// Based on what the server knows about this customer, ask for some
// product recommendations.
request = client.createRequest(INLINE_SERVICE_NAME, "OfferRequest");
request.setSessionKey( SESSION_KEY, sCustID );
IntegrationPointResponseInterface response = client.invoke(request);
```

2. Knowing the number of responses expected allows you handle them accurately. The responses are read from the array and displayed to the customer.

```
if ( response.size() > 0 ){
// Since I know that CrossSell's OfferDecision returns only
// one Choice, I could get that choice from the response with
// response.get(0); Instead, I'll pretend that
// multiple offers could be returned instead of just one.
         System.out.println();
         System.out.println("Here are the deals we've got for you:");
         ResponseItemInterface[] items = response.getResponseItems();
         for ( int i = 0; i < items.length; i++ ){
           System.out.println(" " + (i+1) + ": " + items[i].getId());
           String message = items[i].getValue("message");
           if ( message != null )
             System.out.println("   " + message );
         }
         System.out.println();
         System.out.println("Enter the line number of the offer
         that catches your interest, or zero if none do: " );
```

1.5.9 Closing the loop

Many Inline Services are designed to be self learning. In the Cross Sell Inline Service, the OfferResponse Informant reports interest in a cross sell offer back to a Choice Event model.

```
// Tell the server the good news.
request = client.createRequest(INLINE_SERVICE_NAME, "OfferResponse");
request.setSessionKey( SESSION_KEY, sCustID );
request.setArg( "choiceName", prodName );


// "Interested" is one of the Choice Events defined for the choice
group, Offers.
```

To identify the Choice Event model and Choices see _Knowing_how_to_respond to the serve above.

```
request.setArg( "choiceOutcome", "Interested" );
client.invoke(request);
```

Finally, the session is closed by invoking the CallResolution informant in the server, which in the CrossSell example has been designed to terminate the session.

// Close the server's session.

```
request = client.createRequest(INLINE_SERVICE_NAME, "CallResolution");
request.setSessionKey( SESSION_KEY, sCustID );
client.invoke(request);
```

1.5.10 Closing the Client

When the client application is finished using its `SDClientInterface`, and doesn't intend to use it again, it calls the component's close method, to release any instance-specific information.

```
client.close();
```

Section 2: Using Smart Client JSP tags

A convenient way to integrate a web application to a deployed Inline Service is to use the JSP client integration tags provided. JSP allows you to generate interactive web pages that use embedded Java. The JSP tags provided are based on the Java Smart Client discussed in the previous section.

There is negligible overhead when using the JSP tags. In addition, the tags incorporate automatic reuse of Smart Clients for same session to enhance performance. When a Smart Client is created using the JSP tag, a check is performed to see if a client already exists with the same name and properties and has not been closed. If it does, it automatically reuses that client; if not it will create a new one.

2.1 Integrating with an Inline Service using the JSP Smart Client tags

In general, integration using the Smart Client includes the following steps:

1. Prepare a properties file.
2. Use an Invoke or AsyncInvoke tag to create a request to the server.
3. Gather and parse any response information from Advisors.
4. Close the connection.

2.1.1 JSP Smart Client tag Reference

A reference to the Java Smart Client tags for integration is available through the Siebel Decision Studio online help system.

2.1.2 Preparing the Smart Client example

For this example, the Cross Sell Inline Service has been integrated to a simple command line application to demonstrate how to use the Smart Client for integration.

Use the following steps to prepare the Smart Client example:

1. Copy the file sdclient-test.war from the $INSTALLDIR\Client\Client-Examples directory to the location of your application server under \server\server\default\deploy.

2. In a browser, enter the URL http://localhost:8080/sdclient-test/example.jsp.

2.1.3 Sample JSP code

A working example of using the Smart Client JSP tags for integration can be found at $INSTALLDIR\client\Client Examples\JSP Client Example\example.jsp.

Section 3: .NET Smart Client Integration

The .NET Smart Client provides a very similar to the Java API to make calls from your application. With the current implementation the .NET Smart Client does not have some of the advanced features of the Java Smart Client; however these will be available in the near future.

As these features are transparent to integration using the .NET Smart Client, no additional coding will be needed as the new features are implemented.

The .NET Smart Client is located at $INSTALLDIR`\client\sdclient.dll`. This file should be co-located with your application in order to be accessible.

3.1 Using the .NET Smart Client

The following example outlines how to use the .NET Smart Client for integration to deployed Inline Services on Siebel RTD.

In general, the following are the steps for integration:

1. Create the Siebel RTD Smart Client within your application code.
2. Create a request directed at an Inline Service and an Integration Point.
3. Populate the request with arguments and session keys.
4. Invoke the request using the Smart Client.
5. If the request is invoked on an Advisor, examine the response
6. Close the Smart Client when finished.

3.1.1 .NET API Reference

A reference to the.NET Smart Client API for integration is available through the Siebel Decision Studio online help system.

3.1.2 .NET Integration Example

There is an example of a .NET Integration client in the In the following example Informant and Advisor Integration Points are invoked on the Cross Sell Inline Service. To familiarize yourself with this Inline Service, please see About the Cross Sell Inline Service earlier in this document.

In this simple example, the Integration Points are invoked and the return values from the Advisor are simply written to the console.

Section 4: Zero Client Integration

Siebel Real-Time Decision Server Integration Points are available through a Zero Client approach. Integration Points on a deployed Inline Service are exposed through a Web Services definition.

It is recommended that you work through the tutorial outlined in _Section_2:_How_to use the Java Smar to understand the process of invoking Integration Points.

4.1 About Web Services

The ability to invoke and asynchronously invoke a deployed Integration Point is exposed as a Web Service by the Siebel Real-Time Decision Server. The definition of these operations are available in a WSDL file, located at `$INSTALLDIR\deploy\wireprotocol\wireprotocol.wsdl`. The WSDL file defines all complex types and operations available.

Please note that the WSDL file `$INSTALLDIR\deploy\sdclient\sdclient.wsdl` has been deprecated.