IBM Host Access Transformation Services

Rich Client Platform Programmer's Guide

Version 9.5

SC27-5903-01

IBM Host Access Transformation Services

Rich Client Platform Programmer's Guide

Version 9.5

SC27-5903-01

Note

Before using this information and the product it supports, be sure to read the general information under Appendix B, "Notices," on page 125.

Eighth Edition (November 2015)

Contents

Chapter 1. Introduction

The Host Access Transformation Services (HATS) Toolkit offers many tools for creating and customizing rich client platform (RCP) HATS applications that provide an easy-to-use graphical user interface (GUI) for your character-based 3270 or 5250 host applications. HATS rich client applications can be developed to run in an Eclipse Rich Client Platform (RCP) implementation, in Lotus Notes®, or in the Lotus® Expeditor Client to provide native client applications targeted for an end user's desktop. HATS can also be used to create service-oriented architecture (SOA) assets from logic contained in your character based 3270, 5250, or VT applications.

Because HATS rich client applications are Java™ programs that are deployed as Eclipse plug-ins, you should be familiar with following topics to create your own HATS rich client applications:

- Java programming
- Eclipse plugin development
- Standard Widgets Toolkit (SWT), which is the graphical user interface toolkit used by Eclipse

If not, refer to the following technical documentation:

- Java Tutorials
- Eclipse documentation

You might find that your HATS application requires some additional function that you cannot add using the wizards and editors in HATS Toolkit and IBM® Rational® Software Delivery Platform (SDP). This *Rich Client Platform Programmer's Guide* explains several ways that you can extend your HATS application with additional programming. It also assumes that you are familiar with basic HATS concepts such as:

- How HATS processes host screens
- Creating a transformation using components and widgets
- Events and actions
- Using global variables
- Recording a macro

If you are not already familiar with any of these topics, refer to the information about them in *HATS User's and Administrator's Guide* so that you will have the necessary background to make good use of the information in this book. You should also be familiar with using Rational SDP to create rich client applications.

This *Rich Client Platform Programmer's Guide* describes ways to enhance your HATS application by programming. You can:

- Extend the workbench classes provided by HATS to customize the workbench window where your application is running. Refer to Chapter 2, "Plug-ins and application classes," on page 3 for more information.
- Customize the transformation view of your application, by extending the view class provided in your plug-in project. See "Extending the transformation view's menu" on page 16 for more information.

- Programmatically interact with a HATS session. For example, send a command from another Eclipse view. See “Integration with other Eclipse UI views” on page 41 for more information.
- Add new host components or widgets to be used in transformations by extending the existing host components and widgets. See for more information.
- Customize the perspective of your application. See “The Host Access perspective” on page 13 for more information.

When enhancing your applications, you might find that you need to edit some of the Java source files. Information provided in the section of the Rational Software Delivery Platform help titled *Developing Java applications* can help you with this task.

Code examples

Code examples throughout this guide illustrate the use of the objects or APIs introduced in the adjoining sections. Use these examples in your application only if you are sure that the example code performs the action you intend to perform. The examples may or may not work if you copy them from the book into your application.

Using the API documentation (Javadoc)

The HATS API reference documentation is useful for many programming tasks. To view this documentation, see IBM Knowledge Center collection for HATS at http://www.ibm.com/support/knowledgecenter/SSXKAY_9.5.0 and click the HATS API References (Javadoc) link. Refer to this documentation when you need information about, and examples of, any of the Application Programming Interfaces provided with HATS.

Chapter 2. Plug-ins and application classes

A plug-in is a structured component that describes itself to the *Eclipse platform* using an OSGi manifest (MANIFEST.MF) file and a plug-in manifest (plugin.xml) file. A HATS rich client project is an Eclipse plug-in project that contains artifacts needed for generating a plug-in that runs in an Eclipse environment, such as Lotus Notes or Lotus Expeditor Client. Plug-in projects are developed using the Eclipse Plug-in Development Environment (PDE), which is a component of the Eclipse Software Development Kit (SDK). The Rational Software Delivery Platform (SDP) extends the Eclipse SDK, and HATS further extends the Rational SDP. For more information on the Eclipse PDE, go to http://www.eclipse.org/pde/.

One tool provided by the Eclipse PDE is the plug-in manifest editor. This editor is used to modify the files that describe the plug-in to the Eclipse environment where it is running. Plug-ins generally have two manifest files: plugin.xml and MANIFEST.MF (plugin.xml is optional if the plug-in is not extending another plug-in). If you open either file in the workbench, the same editor is opened, but fields in the editor are tied to different files. The plugin.xml describes how the plug-in extends the platform, what extensions it publishes itself, and how it implements its functionality. The MANIFEST.MF file indicates the ID, version, vendor, and the other plug-ins required by this plug-in.

To edit the plug-in manifest for a HATS rich client project, click the **Open the plug-in manifest** link on the Overview page of the Project Settings editor. The editor will have the following tabs:

Overview	Contains fields for modifying the base attributes of the plug-in, such as such as its plug-in identifier, or ID, and version.
Dependencies	Lists the plug-ins required by this plug-in. If you need to use a function available in another plug-in, use this tab to add a dependency on the plug-in.
Runtime	Displays the list of Java packages available for use by plug-ins that depend on this plug-in.
Extensions	Displays the extension points that are extended by this plug-in.
Extension Points	Displays the extension points defined by a HATS rich client plug-in. By default, no extension points are defined in a HATS rich client plug-in.
Build	Enables you to configure which files are included when this plug-in is built during an export operation. If you add other files to this plug-in, you might need to modify this list. **Notes:** 1. If you add non-Java source files to a plug-in project, you need to update the build.properties file for the plug-in. The build.properties file controls which files are included when a project is exported as a plug-in (first step of

deployment). This file is located at the root of the plug-in project and can be edited using the plug-in manifest editor.

2. If you create a new folder in a plug-in project, the new folder is not included when you export the plug-in unless you add the folder in the **Build** tab.

MANIFEST.MF	Shows the source view of the plug-in MANIFEST.MF file. You should use the **Overview**, **Dependencies,** and **Runtime** tabs to make changes to this file instead of editing the source directly.
plugin.xml	Shows the source view of the plugin.xml file. You should use the **Extensions** and **Extension Points** tabs to make changes to this file.
build.properties	Shows the source of the build.properties file. You should use the **Build** tab to make changes to this file.

Plug-in project extension points

When a plug-in wants to allow other plug-ins to extend or customize portions of its functionality, it will declare an extension point. The extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plug-ins that want to connect to that extension point must implement that contract in their extension. The key attribute is that the plug-in being extended knows nothing about the plug-in that is connecting to it beyond the scope of that extension point contract. This allows plug-ins built by different individuals or companies to interact seamlessly, even without their knowing much about one another.

The com.ibm.hats.rcp.runtime.rcpApplications extension registers your HATS rich client plug-in plug-in with the HATS runtime. This is how the HATS runtime knows which plug-ins are HATS application plug-ins. The Applications view is populated based on plug-ins that implement this extension point. This extension point also indicates which view (by default, your transformation view) is opened when an instance of your application is launched.

The HATS RCP project uses the Eclipse org.eclipse.ui.views extension to register the transformation view class in this plug-in with the Eclipse platform.

Allowing only one instance of an application

By default, a user can work with multiple instances of a HATS rich client application at one time by opening multiple instances of the application's transformation view. To restrict a user from starting multiple instances of an application, you can set a flag in the plug-in manifest file, plugin.xml, for the plug-in project. This flag tells Eclipse to allow only one instance of the application's view. To set this flag:

1. Open the plug-in manifest file for the project by clicking **Open the plug-in manifest** on the **Overview** page of the Project Settings editor.
2. Click the **Extensions** tab.
3. In the **All Extensions** tree, expand the org.eclipse.ui.views node and select the node corresponding to your project. By default, only one view is listed.

4. In the **Extension Element Details** section, change the **allowMultiple** setting from *true* to *false*.
5. Save the file.

Note: Any change to a plug-in manifest file while the application is running requires a restart of the runtime workbench.

At runtime, if a user attempts to open a second instance of the application, the view corresponding to the first application instance receives the focus.

HATS runtime extension plug-in

When you create a HATS rich client plug-in project, a HATS rich client runtime extension plug-in project is also created if one does not already exist in the workspace. This plug-in contains files and settings that are common between all HATS rich client plug-ins running in an environment. This includes, but is not limited to:

- Trace and log settings
- Generated trace and log files
- Registration of key mappings
- Registration of preference pages
- Registration of properties pages

The default name and ID of this plug-in is `com.ibm.hats.rcp.runtime.extension`, and its initial version is `1.0.0`.

Note: Do not change the ID of the HATS RCP runtime extension plug-in. The HATS runtime looks for a plug-in with this ID when it starts.

This plug-in is similar to a HATS EAR project in that it contains files that are common between multiple HATS applications. Unlike a HATS EAR project, only one of these plug-ins can run in an Eclipse environment at one time. This means that trace settings, keyboard mappings, and so forth apply to the entire rich client environment and cannot be configured at the plug-in project level.

Table 1 lists and describes the files included in this plug-in.

Table 1. HATS-specific files in the runtime extension plug-in

File or folder	Description
images (folder)	Contains images for the Applications and Print Jobs views, and an image for an Expeditor application. **Note:** If an image is only used by a single HATS application plug-in, for example, to include an image with a template, consider including it in the application plug-in rather than the runtime extension plug-in.
logs (folder)	Contains HATS trace and messages files.
product.xml	Indicates the version of HATS used to generate this plug-in project. This file is updated when service packs are applied.
runtime.properties	HATS runtime settings, including settings to enable tracing.

Table 1. HATS-specific files in the runtime extension plug-in (continued)

File or folder	Description
runtime-debug.properties	Same settings as runtime.properties, but only used when an Eclipse runtime workbench is launched in debug mode.

Table 2 lists and describes the files included in this plug-in that are not specific to HATS.

Table 2. Files in the runtime extension plug-in not specific to HATS

File or Folder	Description
build.properties	Indicates which files should be packaged in the plug-in when it is exported from the development environment.
MANIFEST.MF	Plug-in descriptor. Indicates ID, name, version of the plug-in, and plug-in dependencies.
plugin.xml	Plug-in descriptor. Indicates the extensions contributed by this plug-in.
plugin_xx.properties	Contains translated strings used by plugin.xml and MANIFEST.MF.

Table 3 lists the extensions contributed by the HATS RCP runtime extension plug-in.

Table 3. RCP runtime extensions

Extension Point ID	Description
org.eclipse.core.runtime.applications	Registers the application class provided in the HATS RCP runtime extension plug-in with the Eclipse platform. See "Application classes" on page 8 for more information.
org.eclipse.core.runtime.products	Defines a product that references the application registered with the org.eclipse.core.runtime.applications extension point. See "Application classes" on page 8 for more information.
org.eclipse.ui.perspectives	Registers the Host Access perspective class provided in the HATS RCP runtime extension plug-in with the Eclipse platform. See Chapter 3, "Perspectives and views," on page 13.
org.eclipse.ui.views	Registers the Applications and Print Jobs views with the Eclipse platform.

Table 3. RCP runtime extensions (continued)

Extension Point ID	Description
org.eclipse.ui.propertyPages	Registers the application properties pages with the Eclipse platform. These pages are displayed when a user right-clicks on an application in the Applications view and selects Properties. You can prevent a page from being available to your users by removing or commenting out its declaration. **Note:** The Connection Parameters Overrides page only appears if connection parameter overriding is enabled in your project. The Variables Override page only appears if global variable overriding is enabled in your project.
org.eclipse.ui.actionSets	Registers the print action used to print what is being displayed on the transformation view.
org.eclipse.ui.preferencePages	Registers the preferences pages provided by HATS. Preferences are usually available by selecting **File > Preferences** or **Window > Preferences** (depending on the Eclipse environment). You can prevent a page from being available to your users by removing or commenting out its declaration. For example, to prevent a user from modifying trace or service settings, you can remove the declared extension for the Troubleshooting page.
org.eclipse.ui.contexts	Registers the Host Access keyboard context. A context defines a group of related commands.
org.eclipse.ui.commands	Registers commands for each of the available actions. A command represents a request from an end user that can be handled by an action.

Table 3. RCP runtime extensions (continued)

Extension Point ID	Description

org.eclipse.ui.bindings	Registers the default keyboard shortcut for a specific command. You can change the default shortcut for a command by modifying the `sequence` attribute of the declaration. A user can override the shortcut for a command using the **Keys** preferences pages.
	
	The key mappings are defined per scheme ID based on the intended runtime client. The schemeId for both Eclipse RCP and Lotus Expeditor Client is:
	`org.eclipse.ui.defaultAcceleratorConfiguration.`
	
	The schemeId for Lotus Notes is:
	`com.ibm.workplace.notes.hannoverConfiguration.`
	
	Note: For examples, see the section, Remapping keys for HATS rich client applications in the *HATS User's and Administrator's Guide*.

com.ibm.eswe.workbench.WctApplication	Registers the Host Access perspective with Lotus Expeditor Client and Lotus Notes. This extension point is used by Lotus Expeditor Client and Lotus Notes to populate the list of applications displayed on the **Open** menu. See Applications in the Lotus Expeditor documentation for more information. **Note:** This declaration does not appear if there are no projects in your workspace targeted for deployment to Lotus Expeditor Client or Lotus Notes.

Application classes

The classes described in this section are responsible for starting and configuring the Eclipse rich client platform environment. For more information on the function and default implementations of these classes, see the Eclipse documentation at http://www.eclipse.org/documentation/, select your version of Eclipse, and search for the section named Building a Rich Client Platform application.

Note: The HostAccessApplication, HostAccessWorkbenchAdvisor, HostAccessWorkbenchWindowAdvisor, and HostAccessActionBarAdvisor classes are only used when Eclipse has been configured, either as part of the product configuration or by startup parameters, to use the application or product defined in the HATS RCP runtime extension plug-in. These classes are not used when this plug-in is running in Lotus Expeditor Client or Lotus Notes since these products use their own application classes.

HostAccessApplication

An Eclipse Application class is the main entry point in an Eclipse environment. When the Eclipse platform starts, it finds and loads the specified application.

A default application class, HostAccessApplication, is provided in the HATS RCP runtime extension plug-in project. This class creates a new workbench window configured by the HostAccessWorkbenchWindowAdvisor class, which is also provided.

Note: Because of the simplicity of the default application class, there is usually no need to modify it.

To use the provided application as the main entry point for an Eclipse environment instance, specify the application's ID in the application argument:

Windows:

```
eclipse.exe -application
com.ibm.hats.rcp.runtime.extension.application
```

Linux:

```
./eclipse -application
com.ibm.hats.rcp.runtime.extension.application
```

An Eclipse product is responsible for branding an Eclipse environment. Branding includes defining an "About" dialog, setting window icons, defining default preferences, and specifying a splash screen which is displayed when the platform starts. For more information on branding, see the Eclipse documentation at http://www.eclipse.org/documentation/, select your version of Eclipse, and search for the section named Customizing a product. A default product is defined in the HATS runtime extension plug-in (com.ibm.hats.rcp.runtime.extension.product). To start the Eclipse platform using this product, specify the product's ID in the product argument:

Windows:

```
eclipse.exe -product com.ibm.hats.rcp.runtime.extension.product
```

Linux:

```
./eclipse -product com.ibm.hats.rcp.runtime.extension.product
```

Because the product extension point indicates which application to use, specifying an application is not required when a product is specified. In the case of the HATS runtime extension plug-in product, the HATS runtime extension plug-in application is used by default.

HostAccessWorkbenchAdvisor

An Eclipse WorkbenchAdvisor class is responsible for specifying the default perspective and constructing a WorkbenchWindowAdvisor object, which is in turn responsible for configuring the workbench window. An implementation of this class is provided for you in the HATS RCP runtime extension plug-in. This class, HostAccessWorkbenchAdvisor, returns a new instance of an HostAccessWorkbenchWindowAdvisor object and specifies `hostaccess.perspectives.main` as the default perspective. This is why the Host Access perspective is displayed by default when a new workbench window is opened. If you have created a custom perspective, you can update the `getInitialWindowPerspectiveId()` method to return the ID of this perspective. Alternatively, you can specify the perspective when Eclipse is launched using the `-perspective` argument:

Windows:

```
eclipse.exe -product com.ibm.hats.rcp.runtime.extension.product
-perspective myCompany.myCustomPerspective
```

Linux:

```
./eclipse -product com.ibm.hats.rcp.runtime.extension.product
-perspective myCompany.myCustomPerspective
```

Note: Prior to Eclipse 3.1, the WorkbenchAdvisor class was responsible for initializing the workbench, configuring the workbench window, and creating actions for the menu bars and toolbars. These responsibilities are now divided between three classes: WindowAdvisor, WorkbenchWindowAdvisor, and ActionBarAdvisor.

HostAccessWorkbenchWindowAdvisor

An Eclipse WorkbenchWindowAdvisor class is responsible for configuring an Eclipse workbench window, including its size, location, and style. An implementation of this class is provided for you in the HATS RCP runtime extension plug-in. This class, HostAccessWorkbenchWindowAdvisor, specifies the initial size of the workbench window and hides the coolbar and status line areas since HATS does not contribute to these areas by default.

This class is also responsible for creating the action bar advisor. An action bar advisor is primarily responsible for configuring the menu bar of the workbench window. See "HostAccessActionBarAdvisor" on page 11 for more information on customizing the menu bar and toolbar of the workbench window.

Controlling the size of the workbench window

Follow these steps to change the default size of the workbench window:

1. From the **Navigator** or **Package Explorer** view, open the HostAccessWorkbenchWindowAdvisor class located in the hostaccess package of the com.ibm.hats.rcp.runtime.extension plug-in project.
2. Find the `preWindowOpen()` method and change the parameter passed to the `setInitializeSize` method of the IWorkbenchWindowConfigurer object. For example, to set the initial window size to 800 pixels wide and 600 pixels tall, update or add the following code:

```
public void preWindowOpen() {
    IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
    configurer.setInitialSize(new Point(800, 600));
    ...
}
```

To automatically maximize the workbench window by default when it is opened, implement the `postWindowCreate()` method of the HostAccessWorkbenchWindowAdvisor class:

```
public void postWindowCreate() {
    PlatformUI.getWorkbench().getActiveWorkbenchWindow().
                    getShell().setMaximized(true);
}
```

Note: You might need to update this class's import section if one or more referenced classes cannot be resolved. To automatically update the import section, right-click inside the editor of the source file and select **Source > Organize Imports**.

Showing the perspective bar

The perspective bar, which resides near the top of the workbench window, provides quick access to perspectives that are already open, and provides the ability for end users to open a new perspective. Showing the perspective bar is a quick and easy way to allow your end users to work with different perspectives. To show the perspective bar:

1. From the **Navigator** or **Package Explorer** view, open the HostAccessWorkbenchWindowAdvisor class located in the hostaccess package of the com.ibm.hats.rcp.runtime.extension plug-in project.

2. Find the `preWindowOpen()` method and update it to call the `setShowPerspectiveBar` method of the IWorkbenchWindowConfigurer object. to `true`:

```
public void preWindowOpen() {
IWorkbenchWindowConfigurer configurer = getWindowConfigurer();
configurer.setShowPerspectiveBar(true);
...
}
```

HostAccessActionBarAdvisor

An Eclipse ActionBarAdvisor class is responsible for configuring the action bars (toolbar, coolbar, menu bar, status line) for a workbench window. An implementation of this class is provided for you in the HATS RCP runtime extension plug-in. This class, HostAccessActionBarAdvisor, creates the actions on the menu bar, including the items under the File menu.

Table 4 lists the methods can be implemented or extended by the HostAccessActionBarAdvisor class.

Table 4. HostAccessActionBarAdvisor Class methods

Method	Description
makeActions(IWorkbenchWindow window)	Action objects should be created and initialized in this method.
fillMenuBar(IMenuManager)	Populates the workbench window's menu bar with menus and submenus. The current implementation of this method creates and populates File, Window, and Help menus. To hide an action that is already showing on the menu, avoid adding it to the MenuManager. For example, to hide the Print action, remove the following line from `fillMenuBar()` method: `fileMenu.add(printAction);` **Note:** The `fillMenuBar()` method is only applicable if the workbench's IWorkbenchWindowConfigurer indicates that the menu bar should be shown.
fillCoolBar(ICoolBarManager)	Populates the workbench window's cool bar with items. **Note:** The `fillCoolBar()` method is only applicable if the workbench's IWorkbenchWindowConfigurer indicates that the cool bar should be shown.

Customizing the workbench window toolbar

To demonstrate how the HostAccessActionBarAdvisor class can be extended, the following steps show how to add a toolbar item to the main coolbar area of the workbench window. When clicked, a new instance of a specified HATS rich client application opens.

1. Update the `HostAccessWorkbenchWindowAdvisor.preOpen()` method to show the workbench window's coolbar area. This method has already been implemented, so you only need to add the call to the `setShowCoolBar(boolean)` method:

```
public void preWindowOpen() {
    ...
    configurer.setShowCoolBar(true);
    ...
}
```

2. Add a new private member to the HostAccessActionBarAdvisor:

```
private LaunchApplicationAction launchAppAction;
```

3. Add code to the `HostAccessActionBarAdvisor.makeActions()` method to create a new instance of the LaunchApplicationAction class. The constructor for this class takes two arguments:
   a. The ID of the HATS rich client application
   b. The IWorkbenchWindow object, which is passed into the `makeActions()` method):

```
protected void makeActions(final IWorkbenchWindow window) {
    ...
    launchAppAction = new LaunchApplicationAction("myPlugin", window);
}
```

4. Add or update the `ApplicationActionBarAdvsior.fillCoolBar()` method to create a toolbar, add the new launcher action to the toolbar, and then add the toolbar to the coolbar:

```
protected void fillCoolBar(ICoolBarManager coolBar) {
    // Create toolbar manager
    IToolBarManager mainToolbar = new ToolBarManager(SWT.FLAT | SWT.RIGHT);
    // Add the launch action to the toolbar manager
    mainToolbar.add(launchAppAction);

    // Add the main toolbar to the window coolbar
    coolBar.add(new ToolBarContributionItem(mainToolbar, "main"));

}
```

5. Launch a new Eclipse workbench. You should have a new toolbar item that, when clicked, will open a new instance of the transformation view associated with this plug-in.

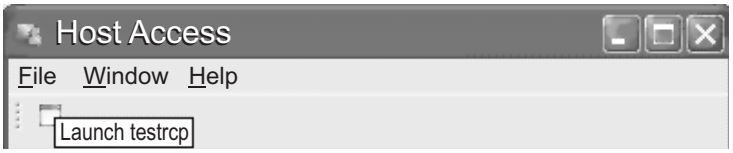


Figure 1. Launching a new workbench

Note: You can change both the default image and text displayed for the launch action by calling the `setText()` and `setImageDescriptor()` methods on the launchAppAction object. See the Eclipse API for the org.eclipse.jface.action.Action class for more information.

Chapter 3. Perspectives and views

A perspective defines the initial layout of views in an Eclipse workbench window. Each perspective provides a set of functions for accomplishing a specific type of task, or works with specific types of resources. For example, the Java perspective combines views commonly used while editing Java source files, and the Debug perspective contains the views for debugging Java programs.

Perspectives control what appears in certain menus and toolbars. Perspectives also define visible action sets, which you can change to customize a perspective. You can save a perspective that you build this way, making a custom perspective that you can open again later.

The Host Access perspective

A default perspective class, hostaccess.perspectives.MainPerspective, has been provided for you in the com.ibm.hats.rcp.runtime.extension plug-in project. This class extends from the HATS com.ibm.hats.rcp.ui.Perspective class which implements the Eclipse perspective interface, org.eclipse.ui.IPerspectiveFactory. The default title for this perspective is **Host Access**, although you can change the title by modifying the plugin_en.properties file in the com.ibm.hats.rcp.runtime.extension plug-in project.

Note: You will need to modify each of the plugin_xx.properties files for each language you will be supporting.

Table 5 lists the methods that can be overridden in your perspective class.

Table 5. Methods to override perspective class

Method	Description
addApplicationsView(IPageLayout)	Adds the HATS applications view to the perspective.
createTransformationViewsPlaceholder(IPageLayout)	Creates a placeholder for the HATS transformation views.
createPrintJobViewsPlaceholder(IPageLayout)</code>	Creates the placeholder area for the 3270 and 5250 print jobs view.
addPrintActionsSet(IPageLayout)</code>	Adds the print action set (defined in the plugin.xml of the com.ibm.hats.rcp.rntime.extension plug-in) to the perspective.

To change the default perspective first opened by the Host Access runtime application, edit the `HostAccessWorkbenchAdvisor` source file in com.ibm.hats.rcp.runtime.extension plug-in project, and change the PERSPECTIVE_ID to your custom perspective ID. See "HostAccessWorkbenchAdvisor" on page 9 for more information about the Host Access perspective.

The following is an example showing how the Host Access perspective is registered in the plug-in descriptor for com.ibm.hats.rcp.runtime.extension:

```
<extension
     point="org.eclipse.ui.perspectives">
     <perspective
            name="%PERSPECTIVE_TITLE"
            icon="images/applications_view.gif"
```

```
            class="hostaccess.perspectives.MainPerspective"
            id="hostaccess.perspectives.main">
      </perspective>
</extension>
```

You are not required to use the Host Access perspective, although it provides your end users with easy access to the HATS applications you have provided them. You can use the Host Access perspective as a starting point, or start from scratch and develop a perspective that better fits the needs of your users.

By default, the Host Access perspective includes the Applications view. See "Applications view" for more information on this view.

See the Javadoc API for more information on the com.ibm.hats.rcp.ui.Perspective class

Note: Lotus Expeditor Client and Lotus Notes developers: Perspectives are not displayed in the Open menu of the Lotus Expeditor Client or Lotus Notes windows. This menu displays the applications that are registered using the com.ibm.eswe.workbench.WctApplication extension point. See "HATS runtime extension plug-in" on page 5 for more information on this extension point.

Applications view

The Applications view (com.ibm.hats.rcp.ui.views.ApplicationsView) is registered in the com.ibm.hats.rcp.runtime.extension plug-in. This view provides users access to the HATS rich client applications installed in the client environment. Users can also configure properties for an application by launching the Properties dialog from this view.

See Applications view in the User's and Administrator's Guide for more information on how to use this view.

Programmatically starting an instance of an application

The Applications view is provided by default on the Host Access perspective so that your end users can start new instances of their HATS rich client applications. An API is provided to enable you to programmatically launch a new instance of a HATS rich client application. This is useful if you have elected to not show the Applications view in your perspective; however, you want to provide a way to launch an application from the perspective or to programmatically launch an application instance from another view. For example, you might have a HATS application to collect data from one host system, launch a new instance of another HATS application, and pass the data to the new instance.

To facilitate passing parameters to the new application instance, the API enables you to override connection parameters and initialize global variable values. To launch a HATS rich client application instance by clicking a button:

1. Use standard SWT programming to add a button and the widgetSelected() code. For more information, see "Editing transformations" on page 19.
2. Right-click the button and select **Events > widgetSelected**.
3. Replace the sample placeholder code in the `widgetSelected(SelectionEvent)` method with the following code:

```
String applicationId = "myApplication";

Properties initParams = new Properties();
initParams.setProperty(com.ibm.eNetwork.beans.HOD.Session.HOST, "129.12.11.2");
initParams.setProperty(com.ibm.eNetwork.beans.HOD.Session.PORT, "623");
initParams.setProperty("hatsgv_userName", "some user");
initParams.setProperty("hatsgv_password", "xxxxx");
initParams.setProperty("hatssharedgv_someVariable", "zzzzz");

ViewInitInfo viewInitInfo = new ViewInitInfo(true, initParams);
RcpUiUtils.launchView(PlatformUI.getWorkbench().getActiveWorkbenchWindow().
                                  getActivePage(), applicationId, viewInitInfo);
```

Note: You must enable connection parameter overriding in the Project Settings of your project to allow any overriding of connection parameters. (You must also enable global variable overriding to allow overriding global variables.) Enabling this function causes the Connection Parameters page to be visible in the properties dialog of the application. See the description of the org.eclipse.ui.propertyPages extension point in Table 3 on page 6 for information on hiding Connection Parameters page using the properties dialog.

Transformation view

A view is a visual component of an Eclipse perspective that displays information to a user. For more information on perspectives, see “The Host Access perspective” on page 13. Each HATS rich client application contributes a view, which extends from the base com.ibm.hats.rcp.ui.views.TransformationView class, to the Eclipse environment it is running in. The transformation view is used by a user to interact with a HATS rich client application.

Table 6 lists the methods of your application’s transformation view class can be overridden.

Table 6. Transformation view methods

Method	Description
`createViewActions()`	Creates actions for the view's menu. **Note:** After actions are created, they must then be registered. See “Extending the transformation view's menu” on page 16 for more information.
`createKeyboardActions()`	Creates keyboard action classes for all known HATS keyboard functions registered in the plugin.xml of the com.ibm.hats.rcp.runtime.extension plug-in project. Once keyboard actions are created, each is registered with the IKeyBindingService of the workbench window by the registerKeyboardActions() method. **Note:** See the User's and Administrator's Guide for information about the `createKeyboardActions()` method and how to add keyboard support for other keys.

Table 6. Transformation view methods (continued)

Method	Description
`getApplicationKeypadDisplayInfo()`	Controls which buttons to display in the application keypad area of the view's toolbar. By default, the application's settings are read to determine what keys to display. You can override this method in your transformation view class if you need to calculate what is displayed. This method has no affect if the view's toolbar is hidden.
`getHostKeypadDisplayInfo()`	Controls which keys to display in the host keypad area of the view, and indicates how the keys are displayed. By default, the application's settings are read to determine which keys to display. You can override this method in your transformation view class if you need to calculate what is displayed.
`getOiaDisplayInfo()`	Indicates what information to display in the OIA of the transformation view. By default, the application's settings are read to determine what to display. You can override this method in your transformation view class if you need to calculate what is displayed.
`getToolbarDisplayInfo()`	Indicates whether the view's toolbar is displayed. If displayed, this method also indicates how buttons are rendered on the toolbar. By default, the application's settings are read to determine how the toolbar is presented. You can override this method in your transformation view class.
`shouldAutoStart()`	Indicates whether the application should connect automatically when this view is opened.

Note: If you need to override how the application keypad, host keypad, OIA, or toolbar is displayed for a specific transformation, override the appropriate method in the transformation class, not in the transformation view class. Only override these methods in the transformation view class if you need to programmatically alter how these areas are displayed. See "Customizing the host keypad" on page 26 for an example of overriding how the host keypad is displayed for a specific transformation.

See the HATS Rich Client API Reference for more information on these methods.

Extending the transformation view's menu

The following code sample shows how to extend your transformation view's menu. In the sample, a disconnect menu item is added which, when clicked, causes the host connection to be disconnected and the disconnect and stop events of the application to be run. You should only add actions to the view's menu that are applicable during the entire lifecycle of the view.

Follow these steps to extend the menu of your transformation view class (the Java source file for this class is located in the <project name>.views package of your project):

1. Since your action needs to be accessed in at least two methods of your transformation view class, declare it as a private member of your transformation view class.

   ```
   private com.ibm.hats.rcp.ui.actions.DisconnectAction disconnectAction;
   ```

2. Override the `createViewActions()` method of the transformation view class, ensuring that you call super.createViewActions() if you want the default actions provided by HATS to be available on the view's menu).

   ```
   protected void createViewActions() {
         super.createViewActions();

         //create an action and append the action to the view's menu.
         disconnectAction = new DisconnectAction("Disconnect",getSessionService());
         getViewSite().getActionBars().getMenuManager().add(new Separator());
         getViewSite().getActionBars().getMenuManager().add(disconnectAction);
      }
   ```

3. `Override the updateViewActions()` method to cause your new action to be enabled or disabled when the menu is updated.

   ```
   protected void updateViewActions() {
         super.updateViewActions();

         //enable the action based on the session state.
         disconnectAction.setEnabled(getSessionService().getSessionServiceState().
                                                         isOperational());
      }
   ```

After adding the Disconnect action, when you click the menu button in the transformation view, you can see Disconnect on the menu:

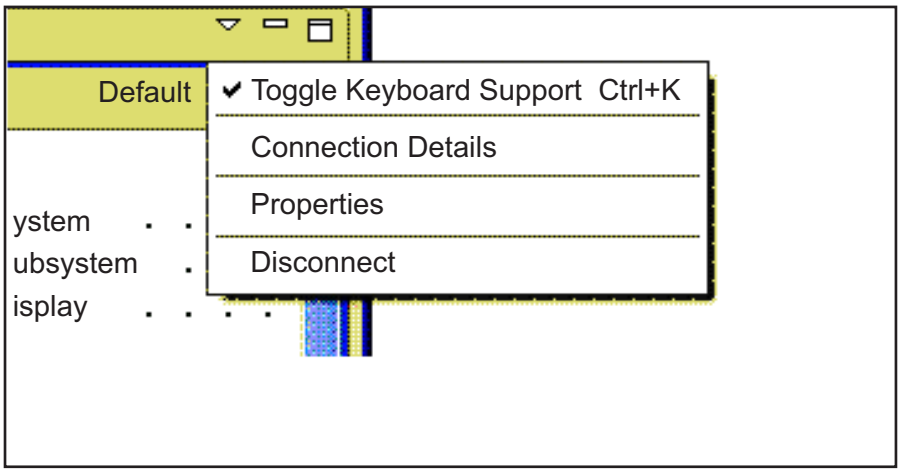


Figure 2. Disconnect on the transformation view menu

Chapter 4. Transformations

A transformation is a HATS resource that controls how a host screen is transformed. In a rich client application, transformations are Java classes that extend from the com.ibm.hats.rcp.ui.transformations.RcpTransformation class. This class extends from the Standard Widget Toolkit (SWT) Composite class, which is a UI control that contains other controls. A transformation can contain SWT widgets, host components (ComponentRendering composites), DefaultRendering composites, global variable controls, macro controls, and host and application keys. Because a transformation is an SWT composite, standard SWT widgets can also be added it.

Some common uses of transformations are:

- Rearranging the presentation of host screen information
- Filtering host screen information that you do not want to show to users
- Presenting host components as SWT widgets in a rich client application.

Editing transformations

Transformations are found in the HATS Projects view under **Rich Client Content > Transformations**.

You can add HATS-specific controls, such as a ComponentRendering composite. Adding a control from the palette generates Java code in the transformation. The generated code references a specific class, as shown in Table 7, based on the selected palette item. The palette items are available in the HATS drawer of the palette:

Table 7. HATS-specific controls and corresponding classes

HATS-specific Palette tools	Class generated in code
Component	com.ibm.hats.rcp.transform.ComponentRendering
Default Rendering	com.ibm.hats.rcp.transform.DefaultRendering
Macro Key	com.ibm.hats.rcp.transform.MacroKey
Global Variable	com.ibm.hats.rcp.transform.GlobalVariableControl
Host Key	com.ibm.hats.rcp.transform.HostKey
Application Key	com.ibm.hats.rcp.transform.ApplicationKey

More information about the classes listed in this table is provided in the following section.

HATS-specific controls

In addition to adding standard SWT controls, you can add HATS-specific controls, such as a ComponentRendering composite, to a transformation. The following sections describe these HATS-specific controls.

The ComponentRendering class

A ComponentRendering composite is a specialized UI control that transforms a given host screen region using a specified component, widget, and settings. Based

on the selected HATS widget, SWT widgets are generated and placed on the ComponentRendering composite. The following code example illustrates the concept:

```
ComponentRendering componentRendering = new ComponentRendering(this, SWT.NONE);
   componentRendering.setScreenCapture(<fully qualified transformation
                                                    class name>.screenCapture);
   componentRendering.setComponent("<fully qualified component class name>");
   componentRendering.setComponentSettings
                           (new com.ibm.hats.common.StringableProperties(""));
   componentRendering.setWidget("<fully qualified widget class name>");
   componentRendering.setWidgetSettings
           (new com.ibm.hats.common.StringableProperties("<widget settings>"));
   componentRendering.setRegion(new com.ibm.hats.transform.regions.
                   BlockScreenRegion(start_row, start_col, end_row, end_col));
   componentRendering.setApplyTextReplacement(true);
   componentRendering.setApplyGlobalRules(true);
```

Note: The `setWidgetSettings` and `setComponentSettings` methods take a standard java.util.Properties object as the parameter. You can construct a Properties object and populate using the `setProperty(String,String)` method, as opposed to using the HATS StringableProperties class.

The DefaultRendering class

A DefaultRendering composite is a specialized UI control that transforms a given host screen region using a rendering set. A rendering set is defined in the **Rendering** tab of the Project Settings editor. Refer to Default Rendering in the *HATS User's and Administrator's Guide* for more information on defining rendering sets. Based on the selected rendering set, SWT widgets are generated and placed on the DefaultRendering composite. The following code example illustrates the concept:

```
DefaultRendering defaultRendering = new DefaultRendering(this, SWT.NONE);
defaultRendering.setScreenCapture(this.screenCapture);
defaultRendering.setRenderingSetName("main");
defaultRendering.setApplyTextReplacement(true);
defaultRendering.setApplyGlobalRules(true);
defaultRendering.setRegion
         (new com.ibm.hats.transform.regions.BlockScreenRegion(1,1,24,80));
```

The MacroKey class

The com.ibm.hats.rcp.transform.MacroKey class represents an individual macro either as a button (org.eclipse.swt.widgets.Button class) or a link (org.eclipse.swt.widgets.Link class). The following code sample shows a MacroKey that is displayed as a button:

```
macroKey = new MacroKey(this, SwtTransformationConstants.BUTTON);
macroKey.setText("macro_1");
macroKey.setMacroName("macro_1");
macroKey.addSelectionListener(new org.eclipse.swt.events.SelectionListener() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        com.ibm.hats.runtime.services.ISessionService
                              sessionService = getSessionService();
        if (sessionService != null) {
            sessionService.playMacro(macroKey.getMacroName());
        }
    }
    public void widgetDefaultSelected(org.eclipse.swt.events.SelectionEvent e) {
    }
});
```

The GlobalVariableControl class

The com.ibm.hats.rcp.transform.GlobalVariableControl class represents a global variable either as a static text (org.eclipse.swt.widgets.Label class) or an input field (org.eclipse.swt.widgets.Text class). The following code sample shows a GlobalVariableControl that displays a global variable as an input field:

```
gvControl = new GlobalVariableControl(this, SwtTransformationConstants.TEXT);
gvControl.setInitialValueFromGlobalVariable(true);
gvControl.setUseAllIndices(true);
gvControl.setGlobalVariableName("userName");
gvControl.setShared(false);
gvControl.setSeparator("");
gvControl.setPasswordField(false);
gvControl.setIndex(0);
gvControl.setGlobalVariableIndexed(false);
```

Note: You can read and change global variables from a transformation without using the GlobalVariableControl class. Please see "Setting and retrieving global variable values" on page 24 for more details.

The HostKey class

The com.ibm.hats.rcp.transform.HostKey class represents an individual host key either as a button (org.eclipse.swt.widgets.Button class) or a link (org.eclipse.swt.widgets.Link class). Multiple keys cannot be included in one HostKey instance. If you need to have two or more keys, define multiple HostKey instances. The following code sample shows a HostKey that is displayed as a button that sends [PF1] to the host when pressed:

```
hostKey = new HostKey(this, SwtTransformationConstants.BUTTON);
hostKey.setText("F1");
hostKey.setCommand("[pf1]");
hostKey.addSelectionListener(new org.eclipse.swt.events.SelectionListener() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        com.ibm.hats.runtime.services.ISessionService
                                    sessionService = getSessionService();
        if (sessionService != null) {
            sessionService.sendCommand(hostKey.getCommand());
        }
    }
    public void widgetDefaultSelected(org.eclipse.swt.events.SelectionEvent e) {
    }
});
```

The ApplicationKey class

This is the class that represents an individual application key either as a Button or a Link. Multiple keys cannot be included in one ApplicationKey class. If you need to have two or more keys, define multiple ApplicationKey instances. The following code sample shows an ApplicationKey that is displayed as a link and sends a disconnect command to the HATS runtime when clicked:

```
applicationKey = new ApplicationKey(this, SwtTransformationConstants.LINK);
applicationKey.setText("<a>Disconnect</a>");
applicationKey.setCommand("disconnect");
applicationKey.addSelectionListener(new org.eclipse.swt.events.SelectionListener() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        com.ibm.hats.runtime.services.ISessionService
                                     sessionService = getSessionService();
        if (sessionService != null) {
            sessionService.sendCommand(applicationKey.getCommand());
        }
```

```
    }
    public void widgetDefaultSelected(org.eclipse.swt.events.SelectionEvent e) {
    }
});
```

Transformation classes

Table 8 lists the methods that can be overridden in a transformation class.

Table 8. RcpTransformation methods

Method	Description
`getApplicationKeypadDisplayInfo()`	Returns an IApplicationKeypadDisplayInfo object that controls how, and if, the application keypad is displayed. This method enables the application keypad to be customized for this transformation. Returning null from this method indicates that the default application keypad, based on the application's default settings, should be displayed.
`getAutoAdvanceOrderer()`	Returns an IAutoAdvanceOrderer object that controls how, and if, auto advance is performed. The default implementation is to return null, indicating that the system should decide the auto advance handler to use from the project settings.
`getDefaultMonospacedFont()`	Allows you to override the default monospaced font at the transformation level. See “Overriding the default monospaced font” on page 27 for an example that uses this method.
`getHostKeypadDisplayInfo()`	Returns an IHostKeypadDisplayInfo object that controls how, and if, the host keypad is displayed. This method enables the host keypad to be customized for this transformation. Returning null from this method indicates that the default host keypad should be displayed based on the application's default settings. Refer to “Customizing the host keypad” on page 26 for examples that use this method.
`getOiaDisplayInfo()`	Returns an IOiaDisplayInfo object that controls how, and if, the OIA is displayed. This method enables the OIA to be customized for this transformation. Returning null from this method indicates that the default OIA should be displayed based on the application's default settings.
`getToolbarDisplayInfo()`	Returns an IToolbarDisplayInfo object that controls how, and if, the toolbar is displayed. This method enables the toolbar to be customized for this transformation. Returning null from this method indicates that the default toolbar should be displayed based on the application's default settings.

Table 8. RcpTransformation methods (continued)

Method	Description
`handlePreSubmit()`	This method is called just prior to a command being submitted. Refer to "Validating input on a transformation" on page 25 for examples that use this method.
`handlePostSubmit()`	This method is called just after a command has been submitted.
`isInitialFocusAtCursorPosition()`	Identifies whether the initial focus is placed at cursor position. For more information, refer to the description of the **Initial cursor position** client setting in the *HATS User's and Administrator's Guide*.
`needsAutoAdvanceSupport()`	Identifies whether the auto advance function for this transformation is enabled. For more information, refer to the description of the **Enable automatic field advance** client setting in the *HATS User's and Administrator's Guide*.
`preRender()`	This method is called prior to the render() method being called.
`postRender()`	This method is called after the render() method is called and the rendering of the transformation is complete.
`setScreenCapture()`	Sets the screen capture for use while editing ComponentRendering and DefaultRendering in the HATS Toolkit. `componentRendering.setScreenCapture( <fully qualified transformation name>.screenCapture);`
`needsArrowKeyNavigationSupport()`	Identifies whether the arrow key navigation support function for this transformation is enabled. For more information refer to the Arrow key navigation support client setting in the *HATS User's and Administrator's Guide*

Samples

Sending a key from a button

Using the palette, add a SWT Button widget to your transformation. Override the `postRender()` method of the RcpTransformation class.

Note: To override the `postRender()` method, follow these steps:

1. Right-click in the Source editor pane.
2. Select **Source > Override/Implement Methods**.
3. Under RcpTransformation, check postRender().
4. Click **OK**.

Add the following code to the `postRender()` method:

```
final RcpContextAttributes attrs = (RcpContextAttributes) getContextAttributes();
  button.addSelectionListener(new SelectionListener()
  {
   public void widgetDefaultSelected(SelectionEvent event) {
```

```
  }

  public void widgetSelected(SelectionEvent event) {
   attrs.getSessionService().sendCommand("[enter]");
  }
 });
```

Updating an input field after the user selects a SWT List widget item

Using the palette, add a SWT List widget to your transformation. Override the `postRender()` method of the RcpTransformation class. Add the following code to the `postRender()` method:

```
//retrieve a model adapter that is bound to a host field at position 1527.
//assuming that the list widget will update the field at this position on the host.
final IModelAdapter modelAdapter = factory.getFieldAdapter(1527);

  list.addSelectionListener(new SelectionListener()
  {
   public void widgetDefaultSelected(SelectionEvent event) {
   }

      //when an item in the list is selected, update the field at position 1527
      //with the value of selected item.
   public void widgetSelected(SelectionEvent event) {
    modelAdapter.setValue(list.getData(list.
                  getItem(list.getSelectionIndex())).toString());
   }
  });
```

Note: If `factory` cannot be resolved, add a declaration for it to the transformation. For example:

```
private HostScreenModelAdapterFactory factory;
```

Setting the value of a global variable from a transformation

The following code sample shows how to prompt a user for a value, in the case the prompt is asking for a customer number, and then set the *customerNumber* global variable with this value.

```
Button button = new Button(this, SWT.NONE);
button.setText("Prompt for command");
button.addSelectionListener(new org.eclipse.swt.events.SelectionAdapter() {

public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
    InputDialog dialog = new InputDialog(getShell(), "Prompt",
                                         "Enter a customer number:", "", null);
    if (dialog.open() == InputDialog.OK) {
        String value = dialog.getValue();
        getSessionService().getParameterDataAccessService().
                setParameterValue("hatsgv_customerNumber",value);
    }
}
 });
```

Note: The HATS runtime is unaware of the new global variable value, set by the `setParameterValue()` method call, until the form is submitted.

Setting and retrieving global variable values

The following example sets a value to the non-shared global variable named “myGlobalVariable”:

```
getSessionService().getParameterDataAccessService().setParameterValue
                               ("hatsgv_myGlobalVariable", "some new value");
```

The following example retrieves a value from the non-shared global variable with the same name.

```
// This will return some string or null
String str = getSessionService().getParameterDataAccessService().
                                 getParameterValue("hatsgv_myGlobalVariable");
```

Note: If the global variable is shared, replace `hatsgv_myGlobalVariable` with `hatssharedgv_myGlobalVariable` in the preceding examples.

Validating input on a transformation

All HATS rich client transformations implement the com.ibm.hats.rcp.transform.IPrePostSubmitHandler interface which provides the ability to control what happens immediately before and after a request is made to the HATS runtime when a specific transformation is displayed. The primary function of this interface is to give you the opportunity to validate values supplied on a transformation by a user before a request is made to the HATS runtime. If a problem is found with one or more of the user's responses, you have the ability to cancel the request, alter the parameters, alter the command, and display a message to the user.

Note: Validation should be performed in the `handlePreSubmit` method of your transformation class, since this method is called just before a request is made to the HATS runtime. Once the request has been made, you cannot alter or cancel it.

The following sample shows how to perform validation on input provided by the user. If an invalid value is specified, the submission is cancelled and a message is displayed to the user.

If the `handlePreSubmit` method has not been overridden in your transformation class, following these steps:

1. Right-click on the source of the transformation (in the Java Editor) and select **Source > Override/Implement Methods**
2. Expand the RcpTransformation node and place a check next to `handlePreSubmit(CommandEvent)` and click **OK**.
3. A stub method is inserted into the transformation source (based on your Java code style preferences, your sample code may not look like the following):

```
public void handlePreSubmit(CommandEvent event) {
    // TODO Auto-generated method stub
    super.handlePreSubmit(event);
}
```

The following code sample shows how to perform a range check on a hypothetical part quantity field. If the quantity supplied by the user is not greater than 500, a message is displayed and the submission is cancelled. It is assumed that this field resides at row 4, column 20 of the host screen that is being transformed by this transformation. Replace your existing `handlePreSubmit(CommandEvent)` method with the following:

```
public void handlePreSubmit(CommandEvent event) {
    // Avoid validation if the user is attempting to disconnect the session
    if (event.getCommand().equals(RuntimeConstants.CMD_DISCONNECT)) {
        return;
    }
```

```
        // Retrieve the value of the "quantity" field using the
        // IHostScreenDataAccessService interface
        // (which is accessed by calling the getHostScreenDataAccessService()
        // method on the session service)
        // The getFieldValue() method takes three parameters
        //
        //   Returns the value of host screen field at the specified position.
        //
        //   @param startPos The start position of the field.
        //   @param offset The offset position of the field.
        //   @param length The length of the field.
        //
        //   @return The value of the host screen field.
        //
        // The Component.convertRowColToPos() method converts a row, column value
        // into a linear screen position.
        // "5" indicates the length of the field.
        String quantityString = getSessionService().
                            getHostScreenDataAccessService().
                            getFieldValue(Component.convertRowColToPos(4, 20,
                            getHostScreen().getSizeCols()),0, 5);

        // Convert the numeric string into an int
        int quantity = 0;
        try {
            quantity = Integer.parseInt(quantityString);
        } catch (Exception ex) {
        }

        // Check to ensure the quantity entered is greater than 500
        if (quantity < 500) {
            // Display a message to the user indicating what the problem is
            MessageDialog.openError(getShell(),
                            getSessionService().getApplication().getName(),
                    "You must order at least 500 units.");

            // Cancel the event (this prevents the request from being made)
            event.setCanceled(true);
        }
    }
```

Customizing the host keypad

- The following code sample shows how to hide the host keypad:

```
public IHostKeypadDisplayInfo getHostKeypadDisplayInfo() {
       HostKeypadDisplayInfo displayInfo = new HostKeypadDisplayInfo();
       displayInfo.setKeypadVisible(false);
       return displayInfo;
}
```

  The default implementation of this method returns `null`, indicating to the HATS runtime that the settings defined in the project settings should be used to show (or not show) the host keypad (and to determine which keys to show). By overriding this method and returning a value other than `null`, the HATS runtime will use this `HostKeypadDisplayInfo` object when the transformation is applied during runtime.

- The following code sample shows how to show only the Enter and F1/Help keys on a host keypad:

```
public IHostKeypadDisplayInfo getHostKeypadDisplayInfo() {
// Construct an array of keys to include on the keypad
  KeypadKey[] keysToDisplay = new KeypadKey[] {
                              new KeypadKey("[enter]", "Enter"),
                              new KeypadKey("[pf1]", "Help") };
```

```
  // Construct and return the keypad display info object
    return new HostKeypadDisplayInfo(keysToDisplay, true,
                                     IHostKeypadDisplayInfo.DISPLAY_BUTTON);
  }
```

Customizing the application keypad

The application keypad can be customized at the transformation-level by overriding the `getApplicationKeypadDisplayInfo()` method of the transformation and returning an object of type `IApplicationKeypadDisplayInfo`.

The following code sample shows how the `getApplicationKeypadDisplayInfo()` method can be overwritten to expose "Page down" and "Page up" to the application keypad when this transformation is applied:

```
public IApplicationKeypadDisplayInfo getApplicationKeypadDisplayInfo() {
 // Define the extra keys
  KeypadKey[] extraKeys = new KeypadKey[] {
                          new KeypadKey(ECLConstants.PAGEDWN_STR, "Page Down"),
                          new KeypadKey(ECLConstants.PAGEUP_STR, "Page Up") };

// Construct a new ApplicationKeypadDisplayInfo object using the project-level
// settings and the new keys
   Application app = getSessionService().getApplication();
   Properties appKeypadSettings = app.getDefaultSettings
      (ApplicationKeypadConstants.SETTINGS_ID);
   ApplicationKeypadDisplayInfo info = new ApplicationKeypadDisplayInfo
      (appKeypadSettings, extraKeys);

  return info;
}
```

To customize the application keypad at the application-level, the `getApplicationKeypadDisplayInfo()` method of your transformation view class "MainView" can be overwritten. See “Transformation view” on page 15 for more information.

Overriding the default monospaced font

This code example causes a smaller font size to be used if the host screen has 132 columns, instead of the standard 80:

```
public Font getDefaultMonospacedFont() {
    if (getHostScreen() != null) {
        if (getHostScreen().getSizeCols() == 132) {
            return FontManager.getInstance(getDisplay()).getFont("Courier New", 8, 0);
        }
    }

    return null;
}
```

Integrating other user interface widgets

The samples in this section require that you add the `org.eclipse.emf.common` plug-in to your rich client project's dependency list. To do this:

1. Open the plug-in manifest file for your project.
2. On the **Dependencies** tab, click **Add** and select `org.eclipse.emf.common`.
3. Save the file.

Binding a SWT Slider widget to a host input field

A slider is a control that enables a user to select a numeric value from a range of values. For example, on the OS/400® main menu, you can use the slider to make a selection.

Using the palette, add a SWT Slider widget to your transformation. Override the `postRender()` method of the RcpTransformation class. Add the following code to the `postRender()` method:

```
// The following code creates 2 adapters :  a FieldAdapter that adapts to a
// Host Screen field located at a specific position, an IControlAdapter that
// adapts to a Slider widget, and bind these 2 adapters so that when slider
// selection is changed, its selection value is updated in the model, and
// when the data in the model is changed, the slider selection is updated.
// In this example, we bind to the a host screen field located at
// position (20,007).
    final RcpContextAttributes attrs = (RcpContextAttributes)
                                                    getContextAttributes();
  HostScreenDataModelManager dataModelManager = attrs.
                                           getHostScreenDataModelManager();
  HostScreenModelAdapterFactory factory = (HostScreenModelAdapterFactory)
                                  dataModelManager.getModelAdapterFactory();
  final IModelAdapter modelAdapter = factory.getFieldAdapter(1527);
  IControlAdapter sliderAdapter = new SliderWidgetAdapter(slider,dataModelManager);
  dataModelManager.bindControlToModel(sliderAdapter, modelAdapter);
```

Example of the SliderWidgetAdapter class

```
//This class allows the slider widget to update a host field with its values and
//also update its value from a host field.
public class SliderWidgetAdapter implements IControlAdapter {

 private Slider slider;  //the target slider widget.
 private IDataModelManager dataModelManager; //handles updating values in the model

 public SliderWidgetAdapter(Slider slider,IDataModelManager dataModelManager) {
  this.slider = slider;
  this.dataModelManager = dataModelManager;

  //create a SelectionListener such that when the value of the slider
  //is changed, it updates the host field.
  this.slider.addSelectionListener(new SelectionListener()
  {
   public void widgetDefaultSelected(SelectionEvent event) {
   }

   public void widgetSelected(SelectionEvent event) {
    updateModel();
   }
  });
 }

 //this method calls the updateModel from the IDataModelManager to
 //update the host field.
 private void updateModel() {
  if ( dataModelManager != null )
   dataModelManager.updateModel(this);
 }

 //return the value of slider selection.
 public String getValue() {
  return slider.getSelection() + "";
 }

 //this method updates the slider selection from a value. This method
 // is called when the value of a host field is changed.
 public void setValue(String value) {
  try {
   slider.setSelection(Integer.parseInt(value));
  }
  catch ( NumberFormatException e ) {
```

```
    slider.setSelection(0);
   }
  }
 }
```

Chapter 5. Templates

Like a rich client transformation, a rich client template is a Java class. It extends from the com.ibm.hats.rcp.ui.templates.RcpTemplate class, which extends from org.eclipse.swt.widgets.Composite. A template can contain SWT widgets, macro controls, and host and application keys. Because a template is an SWT composite, non-HATS widgets can be added to the template.

A template cannot contain host components (ComponentRendering composites), default rendering composites, or global variable controls, while a transformation can contain them.

You can create a template by extending one of the predefined templates that HATS provides. A template defines the basic layout and style of an application, such as:

- Foreground and background colors used for transformations
- Fonts to be used for transformations
- Static labels, such as a logo image
- Links, such as to a corporate home page

Table 9 lists the methods that can be overridden by the RcpTemplate class.

Table 9. RcpTemplate methods

Method	Description
`getDefaultFont()`	Returns the default font to be used by widgets.
`getDefaultMonospacedFont()`	Allows you to override the default monospaced font at the template level.
`getDefaultBackgroundColor()`	Returns the default background color for the template.
`getDefaultForegroundColor()`	Returns the default foreground color for the template.
`getColorMapper()`	Returns an IColorMapper object that controls how host colors are mapped to colors on the view.
`getTableColorProvider()`	Returns an ITableColorProvider object that controls what colors are used for table controls.
`getContentContainer()`	Returns the composite that will be the parent of transformations or other composites displayed on the template.

Refer to the HATS Rich Client Platform API reference for more information on these methods.

The HATS runtime is responsible for constructing and disposing of templates. HATS runtime performs the following steps:

1. Constructs a new instance of the specified template class. This class must be a descendent of RcpTemplate, meaning that it must extend from RcpTemplate or from a class that extends from RcpTemplate.

2. Constructs a new instance of the transformation class.
3. Calls the `setContent()` method of the template instance, passing it the transformation instance. This method calls the `applyStyleToComposite()` method of the transformation instance
4. Calls the `setContent()` method of the transformation view instance, passing it the template instance.

Editing templates

Templates in HATS rich projects are Java SWT composites and you can edit them manually using Java Editor or any text editors. You can find the templates in the HATS Projects view under **Rich Client Content > Templates**.

By default, the templates shipped by HATS return the background color, foreground color, and font of the template class for the methods of RcpTemplate. It is recommended that you use the Properties view in the Visual Editor to update the background color, foreground color, and font of the template if you implement the IRcpTemplate interface.

Note: As documented in the Eclipse API reference, you must dispose of Image, Color, and Font objects that you create. You should not dispose of a color retrieved using the `Display.getSystemColor()` method, since you did not create it. For non-standard colors, it is recommended that you use the ColorManager class provided by HATS. This class manages the creation, caching, and disposing of colors. You should not dispose of colors created by the ColorManager. A FontManager class is also provided, which works similarly to the ColorManager class, except that it manages fonts.

Samples

Customizing host color mappings

The mapping of host colors to colors on the view are controlled by the IColorMapper object returned by the `getColorMapper()` method of the template class. The default implementation of this method returns an object of type DefaultColorMapper. The following color mapper classes are provided by HATS:

- com.ibm.hats.rcp.ui.templates.DefaultColorMapper
- com.ibm.hats.rcp.ui.templates.WhiteBackgroundColorMapper

DefaultColorMapper should be used by templates that have a non-white background color since it maps the color of white host fields to white. WhiteBackgroundColorMapper should be used by templates that have a white or light background color since it maps the color of white host fields to black and uses darker colors (compared to the default color mapper).

The `mapColor()` method of the IColorMapper object is responsible for mapping a host color to an SWT RGB value. This method is typically called from the HATS field widget when the **Enable foreground colors** setting is enabled. Table 10 lists the possible host color values that can be supplied to this method.

Table 10. Colors supplied to the mapColor method

Host Color	Description
0	Blank
1	Blue

Table 10. Colors supplied to the mapColor method (continued)

Host Color	Description
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown (3270), Yellow (5250)
7	White (normal intensity)
8	Gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	White (high intensity)

There are two approaches for altering how colors are mapped by your template:

1. Create a class that implements the IColorMapper interface, implement the `mapColor()` method, and update your template to return a new instance of this class
2. Create a class that extends from one of the provided color mapper classes, override the `mapColor()` method, and update your template to return a new instance of this class.

If you choose to extend one of the classes provided by HATS, you will need to override the `mapColor()` method. The following code sample shows how to change the color for blue (host color 1) and magenta (host color 5) host fields:

```
public class MyCustomColorMapper extends DefaultColorMapper {

 public RGB mapColor(int hostColor) {
  RGB rgb = null;

  if (hostColor == 1) {
   rgb = new RGB(100, 100, 100);
  } else if (hostColor == 5) {
   rgb = new RGB(255, 0, 0);
  } else {
   rgb = super.mapColor(hostColor);
  }

  return rgb;
 }

}
```

The following code sample shows how to implement the `getColorMapper()` method of your template class to return your custom color mapper.

Notes:

1. This method might already exist in your template. If so, replace the current method with the following:

```
public IColorMapper getColorMapper() {
        return new MyCustomColorMapper();
    }
```

2. Color mappings are controlled at the template level. If you need to alter how colors are mapped at the transformation level, you need to create a new template, which can extend your existing template, and configure your screen customizations to use this template.

Removing borders from input fields

To remove borders from input fields rendered by HATS widgets, you can override the `getControlStyleClass(Class)` method of your template and return 0 when the specific class is the SWT Text widget class (org.eclipse.swt.widgets.Text).

To override the `getControlClassStyle()` method, do the following:

1. Click in the **Source** editor pane.
2. Click Select **Source > Override/Implement Methods**.
3. Under **RcpTemplate**, select the **getControlClassStyle** checkbox.
4. Click **Ok**.

Add the following code to the `getControlClassStyle()` method:

```
if (controlClass.equals(org.eclipse.swt.widgets.Text.class)) {
  return 0;
 } else {
  return super.getControlClassStyle(controlClass);
 }
```

Chapter 6. Runtime services

A set of services, collectively called the HATS Runtime Services, are provided as an interface to the HATS runtime, HATS rich client applications, and running application instances. Using this set of provided APIs is the recommended method of interaction with the HATS runtime, since you don't need to be concerned with the underlying details.

Table 11 describes the types of services provided. There is only one instance of the runtime service and client service per Eclipse environment, but there can be multiple instances of the application service (since multiple HATS rich client applications can be installed into an Eclipse environment) and the session service (since a user can start multiple sessions, or application instances).

Table 11. Provided services

Service	Description	Scope
Runtime	Provides runtime-related services, such as runtime initialization, creating new client sessions, providing access to the connection manager and application manager classes.	One per Eclipse environment.
Application	Provides interface for retrieving information about the HATS rich client application.	One per HATS rich client plug-in application.
Client	Provides interface for retrieving information about a client or user.	One per Eclipse environment.
Session	Provides interface for interacting with HATS runtime on behalf of a particular application instance / host connection.	One per running application instance.

Note: Service classes should not be extended.

Note: For code examples in the following sections, the following imports might be needed:

```
import java.util.Collection; import java.util.Iterator;
import com.ibm.hats.rcp.runtime.RcpRuntimePlugin;
import com.ibm.hats.runtime.services.IApplicationService;
import com.ibm.hats.runtime.services.IClientService;
import com.ibm.hats.runtime.services.IRuntimeService;
import com.ibm.hats.runtime.services.IServiceManager;
import com.ibm.hats.runtime.services.ISessionService;
import com.ibm.hats.runtime.services.ServiceType;
```

Accessing the service manager

All services are created and maintained by a single service manager (which implements the com.ibm.hats.runtime.services.IServiceManager interface). Service objects should not be constructed outside of the service manager. The following code sample shows how to access the service manager:

```
IServiceManager serviceManager = RcpRuntimePlugin.getDefault().getServiceManager();
```

Table 12 identifies the methods that can be called by the IServiceManager object.

Table 12. IServiceManager methods

Method	Description
`addServiceManagerListener(ServiceManagerListener)`	Adds a listener to this service manager.
`removeServiceManagerListener(ServiceManagerListener)`	Removes a listener from this service manager.
`getApplicationService(String)`	Returns the IApplicationService corresponding to the specified application plug-in ID.
`getClientService(String)`	Returns the IClientService corresponding to the specified client ID (in the rich client, the client ID is always the value of RcpRuntimeService.rcpClientId).
`getRuntimeService()`	Returns the IRuntimeService for the environment.
`getSessionService(String, String, String)`	Returns the ISessionService corresponding to the specified client ID, application plug-in ID, and view ID.
`getServiceIDs(ServiceType)`	Returns a set of IDs corresponding to services managed by this service manager with the specified service type getServiceEntryCount(ServiceType).
`getServiceEntryCount(ServiceType)`	Returns the number of services managed by this service manager with the specified type.

See the com.ibm.hats.runtime.services.IServiceManager API for more information.

Using the runtime service

The runtime service, which implements the com.ibm.hats.runtime.services.IRuntimeService interface, provides methods for interacting with the HATS runtime. Most of the methods provided on this service should not be called directly, but are made public so other components of the HATS runtime can communicate. The runtime service object for an application should be retrieved from the service manager. Sample code to retrieve the runtime service:

```
IRuntimeService runtimeService =
            serviceManager.getRuntimeService();
```

The following methods can be called by the runtime service:

Table 13. IRuntimeService methods

Method	Description
`getConnectionManager()`	Returns the default ConnMgr instance.
`getServiceManager()`	Returns the service manager that instantiated this service.

Using the application service

The application service, which implements the com.ibm.hats.runtime.services.IApplicationService interface, provides methods for retrieving information related to a HATS rich client application. Each HATS rich client application plug-in installed and enabled in the Eclipse environment will have an associated IApplicationService object. The application service object for an application should be retrieved from the service manager. Sample code to retrieve the application service:

```
IApplicationService applicationService =
                serviceManager.getApplicationService("myPluginId");
```

The `getApplicationService(String)` method will return null if the specified application ID is invalid or the plug-in failed to start.

Table 14 identifies the methods that can be called by the IApplicationService object.

Table 14. IApplicationService methods

Method	Description
`getApplication()`	Returns an Application object which contains information about the application. This object is not guaranteed to be synchronized with the application.hap file, and the use of the object returned by this method should only be used to retrieve information. Do not use this object to alter any settings.
`getApplicationId()`	Returns the application identifier.
`getConfig()`	Returns the configuration.
`getRuntimeService()`	Returns the runtime service.
`getServiceManager()`	Returns the service manager that instantiated this service.

Using the client service

The client service, which implements the com.ibm.hats.runtime.services.IClientService interface, provides methods for retrieving information related to the client service. Sample code to retrieve the client service:

```
IServiceManager serviceManager = RcpRuntimePlugin.getDefault().getServiceManager();
IClientService clientService = serviceManager.getClientService("theClientId");
```

Table 15 on page 38 identifies the methods that can be called by the IClientService object.

Table 15. IClientService methods

Method	Description
`getClientId()`	Returns the client identifier, or ID. In the rich client, this is always the value of *RcpRuntimeService.rcpClientId*.
`getRuntimeService()`	Returns the runtime service.
`getServiceManager()`	Returns the service manager that instantiated this service.
`getSession(String)`	Returns the specified session. **Note:** This is the host session of type ISession, not the session service instance.

Using the session service

The session service, which implements the com.ibm.hats.runtime.services.SessionService interface, provides methods for interacting with an instance of a HATS rich client application. Each running application instance, for example, a transformation view instance, has an associated session service. There is a one-to-one mapping between a transformation view instance and a session service.

Sample code for retrieving the complete collection of session services:

```
IServiceManager serviceManager =
         RcpRuntimePlugin.getDefault().getServiceManager();
Collection sessionServices =
         serviceManager.retrieveServiceEntries( ServiceType.SESSION );
Iterator itSessionServices =
         sessionServices.iterator();
while ( itSessionServices.hasNext() )
{
ISessionService aSessionService =
      (ISessionService) itSessionServices.next();
// Your custom code goes here
}
```

Table 16 identifies the methods that can be called by the ISessionService object representing an instance of an application.

Table 16. ISessionService methods

Method	Description
`addPresentationListener (IPresentationListener)`	Adds the specified presentation listener to the list. Listeners are notified when the presentation is updated.
`addSessionServiceListener (ISessionServiceListener)`	Adds the specified session service listener to the list. Listeners are notified when the state of the session has changed or when a command is about to be sent.
`canSendCommand(String)`	Returns a boolean value of true if the current session service state can send the specified command, otherwise returns a boolean value of false.

Table 16. ISessionService methods (continued)

Method	Description
`disconnect()`	Disconnects the host session. This method is called by the Disconnect button on the Application keypad. The associated transformation view is not closed by this method. This method will perform no action if the session service is already processing a command.
`getApplication()`	Returns the application associated with this session service instance. The use of the object returned by this method should only be used to retrieve information. Do not use this object to alter any settings.
`getApplicationId()`	Returns the application ID (for example, the plug-in ID) of the application associated with this session service.
`getApplicationService()`	Returns the application service associated with this session service instance.
`getClientId()`	Returns the client ID associated with this session service instance.
`getCurrentState()`	Returns the current state of this session service instance.
`getHostScreen()`	Returns the host screen associated with this session service instance. This returns the last host screen processed by the HATS runtime, not necessarily the current host screen.
`getHostScreenDataAccessService()`	Returns the model containing the values of host field-associated controls on the form. This method is only applicable when a transformation is currently being displayed on the view.
`getLocale()`	Returns the locale that should be used to display messages. This locale might be different than the locale of the Eclipse environment if you have changed the client locale settings in the Project Settings editor for the application.
`getMacroPromptDataAccessService()`	Returns the service used to access the model data.
`getParameterDataAccessService()`	Returns the model containing the values of non-host field-associated controls on the form. This service is used to set and retrieve other form values, such as global variable values, that will be passed to the HATS runtime when the current form is submitted.
`getRuntimeService()`	Returns the runtime service.
`getServiceManager()`	Returns the service manager that instantiated this service.
`getSession()`	Returns the host session associated with this session service instance.
`getSessionServiceState()`	Returns the session service state as a state object.
`getViewId()`	Returns the transformation view instance ID.

Table 16. ISessionService methods (continued)

Method	Description
`isAsyncUpdateEnabled()`	Returns a boolean value of true if asynchronous update is enabled; otherwise returns a boolean value of false.
`isCommandSupported(String)`	Returns a boolean value of true if the specified command is supported; otherwise returns a boolean value of false.
`playMacro(String)`	Plays the specified macro.
`playMacro(String, Properties)`	Plays the specified macro and passes values for any prompts defined in the macro.
`removePresentationListener (IPresentationListener)`	Removes the specified presentation listener from the list.
`removeSessionServiceListener (ISessionServiceListener)`	Removes the specified session service listener from the list.
`sendCommand(String)`	Sends the specified command. Common commands are: [pf1], [pf2], [enter], [fldext]. **Note:** The square brackets around some commands are required.
`sendCommand(String, Properties)`	Sends the command specified by String along with the associated properties. A command is either a Host On-Demand mnemonic, such as [enter], or a HATS command, such as default. Additional parameters can be specified. These parameters are included in the request that gets submitted. These parameters override any parameters of the same name that are collected from the SDO model, such as host input fields, or global variables. Override parameter values should be of type String[].
`sendContinue()`	Sends a **continue** command. This command must be sent to continue from a show composite action or a macro prompt action.
`sendEnter()`	Sends the Enter key command.
`sendF1()` - `sendF24()`	Sends the respective PF key command.
`sendRefresh()`	Sends the Refresh command.
`sendShowDefault()`	Sends the Show default command.
`start(Properties)`	Start the session service using the specified connection and global variable overrides.
`stop()`	Stops the session service.
`updatePresentation(IPresentable)`	Updates the presentation (generally the transformation view) with the specified presentable.

The associated ISessionService object for a transformation view can be retrieved by calling the `getSessionService()` method on the view. This method is provided by the ITransformationView interface, which is implemented by the base TransformationView class.

Integration with other Eclipse UI views

By using the Runtime Services APIs, HATS transformation views, or HATS rich client applications, can communicate with other Eclipse UI views, or non HATS rich client applications. This communication can be either incoming or outgoing with respect to the HATS application. Incoming communication is defined as a HATS rich client session receiving data or commands from an entity external to HATS. Outgoing communication is defined as a HATS rich client session sending data or commands to an external entity. This functionality enables pertinent data to be retrieved from multiple sources and displayed in a single view. Therefore, the HATS rich client application can either be the view receiving and displaying data from other Eclipse rich client applications or the application providing data to one or more Eclipse rich client applications for display.

An incoming communication scenario

In the following scenario, assume that you have a host application that takes a flight number for input and displays information about that flight. This application has been transformed into a HATS RCP application. Another Eclipse RCP-based application takes reservation information including the flight number. When the flight number is entered into this application, it can be transmitted to the HATS RCP application causing the HATS view to be updated with information for that flight number. The following is a code example for this scenario:

```
IServiceManager serviceManager = RcpRuntimePlugin.getDefault().getServiceManager();
IClientService clientService = serviceManager.getClientService("theClientId");

/*---------------------------------------------------------------------------
//
// Provides an example of a HATS application receiving information from another
// application.  This example cannot be executed directly but is provided to
// highlight the approach required to achieve this functionality.
//
// @param  Flight number from GUI as a Java String.  This value cannot be
// <code>null</code>.
//
*///-------------------------------------------------------------------------
    public void inboundToHATSApplication( String flightNumber )
    {
        // Save flight number from GUI
        Properties params = new Properties();

        params.setProperty( "fltNum",  flightNumber );

        // Get the service manager from singleton plug-in instance.
        IServiceManager serviceManager =
                                RcpRuntimePlugin.getDefault().getServiceManager();

        // This will be the plug-in session we use to play our macro
        ISessionService aSessionService = null;

        // Find the correct HATS plug-in session
        Collection sessionServices =
                    serviceManager.retrieveServiceEntries( ServiceType.SESSION );
        Iterator itSessionServices = sessionServices.iterator();

        while ( itSessionServices.hasNext() )
        {
            aSessionService = (ISessionService) itSessionServices.next();

            // Assumes the first instance of the HATS plug-in called "myApp"
            // is the one we want.
            if ( "myApp".getApplicationId().equals( aSessionService ) )
            {
```

```
                break;
            }
        }

        // Refer to documentation to see how to launch an RCP plug-in instance if
        // one is not already launched
        if ( aSessionService != null )
        {
            // Really should check the state in a limited-time loop,
            // but let's assume we can go ahead
            if ( aSessionService.getSessionServiceState().isOperational() )
            {
                // Ask HATS session service to play the macro with the fltNum
                // from our GUI
                // Assumes that the macro can start from current screen or from
                // the results screen.
                aSessionService.playMacro( "displayFlightInfo", params );
            }
        }
    }
```

Samples

Sample class and methods showing how to access the different runtime services

```
/*--------------------------------------------------------------------------------
//
// Host Access Transformation Services technology
//
// Module Name: RuntimeServicesExamples.java
//
// Description: Provides java code samples for accessing HATS Rich Client Program
// Runtime Services.
//
*///------------------------------------------------------------------------------

package com.ibm.hats;

import java.util.Collection;
import java.util.Iterator;

import com.ibm.hats.rcp.runtime.RcpRuntimePlugin;

import com.ibm.hats.runtime.services.IApplicationService;
import com.ibm.hats.runtime.services.IClientService;
import com.ibm.hats.runtime.services.IRuntimeService;
import com.ibm.hats.runtime.services.IServiceManager;
import com.ibm.hats.runtime.services.ISessionService;
import com.ibm.hats.runtime.services.ServiceType;

//------------------------------------------------------------------------------


/** <code><B>
*  Class Name:  </B> RuntimeServicesExamples <B><BR>
*  Class Type:  </B> Normal <B><BR>
*  Base Class:  </B> None   <B><BR>
*  Intf Class:  </B> None   <B><BR>
*  Description: </code></B> Rich Client Program Runtime Services java code samples.
*///------------------------------------------------------------------------------
public class RuntimeServicesExamples
{
    //----------------------------| Constants |---------------------------------
```

```
//----------------------------| Variables |---------------------------------

//-----------------------------------------------------------------------------
/**
* The default constructor.
*///---------------------------------------------------------------------------
public RuntimeServicesExamples()
{
    super();
}

//-----------------------------------------------------------------------------
/**
* Provides an example of accessing each application service.
*///---------------------------------------------------------------------------
public void accessApplicationService()
{
    Collection applicationServices = this.retrieveApplicationServices();

    Iterator itApplicationServices = applicationServices.iterator();

    while ( itApplicationServices.hasNext() )
    {
        IApplicationService aApplicationService = (IApplicationService)
                                          itApplicationServices.next();

        // Custom code goes here
    }
}

//-----------------------------------------------------------------------------
/**
* Provides an example of accessing each client service.  Currently, there is
// only one.
*///---------------------------------------------------------------------------
public void accessClientService()
{
    Collection clientServices = this.retrieveClientServices();

    Iterator itClientServices = clientServices.iterator();

    while ( itClientServices.hasNext() )
    {
        IClientService aClientService = (IClientService)
                                                    itClientServices.next();

        // Custom code goes here
    }
}

//-----------------------------------------------------------------------------
/**
* Provides an example of accessing each runtime service.  Currently, there is
// only one.
*///---------------------------------------------------------------------------
public void accessRuntimeService()
{
    Collection runtimeServices = this.retrieveRuntimeServices();

    Iterator itRuntimeServices = runtimeServices.iterator();

    while ( itRuntimeServices.hasNext() )
    {
        IRuntimeService aRuntimeService = (IRuntimeService)
                                                    itRuntimeServices.next();

        // Custom code goes here
```

```
    }
}

//-------------------------------------------------------------------------
/**
* Provides an example of accessing each session service.
*///-------------------------------------------------------------------------
public void accessSessionService()
{
    Collection sessionServices = this.retrieveSessionServices();

    Iterator itSessionServices = sessionServices.iterator();

    while ( itSessionServices.hasNext() )
    {
        ISessionService aSessionService = (ISessionService)
                                                    itSessionServices.next();

        // Custom code goes here
    }
}

//-------------------------------------------------------------------------
/**
* Retrieve the client service.
*
* @return  The client service.
*///-------------------------------------------------------------------------
public IClientService getClientService( final String clientId )
{
    return this.getServiceManager().getClientService( clientId );
}

//-------------------------------------------------------------------------
/**
* Retrieve the runtime service.
*
* @return  The runtime service.
*///-------------------------------------------------------------------------
public IRuntimeService getRuntimeService()
{
    return this.getServiceManager().getRuntimeService();
}

//-------------------------------------------------------------------------
/**
* Retrieve the service manager.
*
* @return  The service manager.
*///-------------------------------------------------------------------------
public IServiceManager getServiceManager()
{
    return RcpRuntimePlugin.getDefault().getServiceManager();
}

//-------------------------------------------------------------------------
/**
* Provides an example of retrieving a Collection of current application
* service objects.
*
* @return  A Collection of current application service objects.
*///-------------------------------------------------------------------------
public Collection retrieveApplicationServices()
{
    IServiceManager manager = this.getServiceManager();

    Collection applicationServices = manager.retrieveServiceEntries(
```

```
                                                    ServiceType.APPLICATION );

    return applicationServices;
}

//-----------------------------------------------------------------------------
/**
* Provides an example of retrieving a Collection of current client service
* objects.  Currently, there is only one.
*
* @return  A Collection of current client service objects.
*///---------------------------------------------------------------------------
public Collection retrieveClientServices()
{
    IServiceManager manager = this.getServiceManager();

    Collection clientServices = manager.retrieveServiceEntries(
                                                        ServiceType.CLIENT );

    return clientServices;
}

//-----------------------------------------------------------------------------
/**
* Provides an example of retrieving a Collection of current runtime service
* objects.  Currently, there is only one.
*
* @return  A Collection of current runtime service objects.
*///---------------------------------------------------------------------------
public Collection retrieveRuntimeServices()
{
    IServiceManager manager = this.getServiceManager();

    Collection runtimeServices = manager.retrieveServiceEntries(
                                                        ServiceType.RUNTIME );

    return runtimeServices;
}

//-----------------------------------------------------------------------------
/**
* Provides an example of retrieving a Collection of current session service
* objects.
*
* @return  A Collection of current session service objects.
*///---------------------------------------------------------------------------
public Collection retrieveSessionServices()
{
    IServiceManager manager = this.getServiceManager();

    Collection sessionServices = manager.retrieveServiceEntries(
                                                        ServiceType.SESSION );

    return sessionServices;
}
//-----------------------------------------------------------------------------
/**
* Provides an example of turning off border rendering for all input fields.
*
*
*///---------------------------------------------------------------------------
public int getControlClassStyle(class controlClass)
{
  if (controlClass.equals(Text.class))
  {
    return 0;// super.getControlClassStyle(controlClass);
   }
```

```
      else
      {
        return super.getControlClassStyle(controlClass);
    }
}
```

Listening for 3270 Print Jobs

The following example shows how you can use the service manager to enable your plugin to listen for 3270 print job events. These changes are made to the RCP application plugin class:

1. Implement `ServiceManagerListener`, `ISessionServiceListener`, and `IPrintJobManagerChangeListener`
2. Implement the `ServiceManagerListener` method `public void serviceChanged(ServiceManagerEvent event)`. This method is called when the service manager changes. In this example, it is used to determine when a new session has been created or destroyed. When a new session is created, add yourself as a listener. When a session is destroyed, remove yourself as a listener.
3. Implement the `ISessionServiceListener` methods
   - `public void sessionStateChanged(StateChangeEvent event)` - This method is called when the state of the session changes. Use this method to find the sessions print job manager and add yourself as a listener to it.
   - `public void aboutToProcessCommand(CommandEvent event)` - This method is left empty for this example.
   - `public void afterProcessCommand(CommandEvent event)` - This method is left empty for this example.
4. Implement the IPrintJobManagerChangeListeners
   - `public void addPrintJob(PrintJobManager printJobManager, PrintJob printJob)` This method is called when a print job has been added to the print job manager.
   - `public void removePrintJob(PrintJobManager printJobManager, PrintJob printJob)` This method is called when a print job is removed from the print job manger (deleted).
   - `public void updatePrintJob(PrintJobManager printJobManager, PrintJob printJob)` This method is called when a print job is updated.
5. Add yourself as a service manager listener in the `start()` method of your application plug-in.
6. Remove yourself as a service manager listener in the `stop()` method of your application plug-in.

```
package printExample;

import java.util.HashSet;

import org.eclipse.jface.resource.ImageDescriptor;
import org.osgi.framework.BundleContext;

import com.ibm.hats.rcp.runtime.RcpRuntimePlugin;
import com.ibm.hats.rcp.ui.AbstractRcpApplicationPlugin;
import com.ibm.hats.runtime.ApplicationSpecificInfo;
import com.ibm.hats.runtime.ClientSpecificInfo;
import com.ibm.hats.runtime.IPrintJobManagerChangeListener;
import com.ibm.hats.runtime.PrintJob;
import com.ibm.hats.runtime.PrintJobManager;
import com.ibm.hats.runtime.PrintResourceHandler;
import com.ibm.hats.runtime.PrintSpecificInfo;
import com.ibm.hats.runtime.events.CommandEvent;
```

```
import com.ibm.hats.runtime.events.ISessionServiceListener;
import com.ibm.hats.runtime.events.ServiceManagerEvent;
import com.ibm.hats.runtime.events.ServiceManagerListener;
import com.ibm.hats.runtime.events.StateChangeEvent;
import com.ibm.hats.runtime.services.IService;
import com.ibm.hats.runtime.services.IServiceManager;
import com.ibm.hats.runtime.services.ISessionService;
import com.ibm.hats.util.StateType;

/**
 * The activator class controls the plug-in life cycle
 */
public class PrintExamplePlugin extends AbstractRcpApplicationPlugin implements
                               ServiceManagerListener, ISessionServiceListener,
                               IPrintJobManagerChangeListener
{
    // The plug-in ID
    public static final String PLUGIN_ID = "PrintExample";

    // The shared instance
    private static PrintExamplePlugin plugin;

    // The service manager
    private IServiceManager serviceManager = null;

    // The print managers we are listeners for
    private HashSet printJobManagers = new HashSet();

    /**
     * The constructor
     */
    public PrintExamplePlugin()
    {
        plugin = this;

        System.out.println("PrintExamplePlugin: ctor");
    }

    /*
     * (non-Javadoc)
     *
     * @see org.eclipse.ui.plugin.AbstractRcpApplicationPlugin#
     *                  start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception
    {
        super.start(context);

        // Add the service manager listener
        serviceManager = RcpRuntimePlugin.getDefault().getServiceManager();
        serviceManager.addServiceManagerListener(this);
    }

    /*
     * (non-Javadoc)
     *
     * @see org.eclipse.ui.plugin.AbstractRcpApplicationPlugin#stop(org.osgi.
     *                                                  framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception
    {
        plugin = null;

        // Remove the service manager listener
        if (serviceManager != null)
            serviceManager.removeServiceManagerListener(this);
```

```
        super.stop(context);

    }

    /**
     * Returns the shared instance
     *
     * @return the shared instance
     */
    public static PrintExamplePlugin getDefault()
    {
        return plugin;
    }

    /**
     * Returns an image descriptor for the image file at the given plug-in
     * relative path
     *
     * @param path
     *            the path
     * @return the image descriptor
     */
    public static ImageDescriptor getImageDescriptor(String path)
    {
        return imageDescriptorFromPlugin(PLUGIN_ID, path);
    }

    /*
     * (non-Javadoc)
     *
     * @see com.ibm.hats.runtime.IPrintJobManagerChangeListener#
     *         addPrintJob(com.ibm.hats.runtime.PrintJobManager,
     *         com.ibm.hats.runtime.PrintJob)
     */
    public void addPrintJob(PrintJobManager printJobManager,
                                            PrintJob printJob)
    {
        System.out.println("PrintExamplePlugin: addPrintJob:
                                      " + printJob.toString());
    }

    /*
     * (non-Javadoc)
     *
     * @see com.ibm.hats.runtime.IPrintJobManagerChangeListener#
     *      removePrintJob(com.ibm.hats.runtime.PrintJobManager,
     *      com.ibm.hats.runtime.PrintJob)
     */
    public void removePrintJob(PrintJobManager printJobManager,
                                               PrintJob printJob)
    {
        System.out.println("PrintExamplePlugin: removePrintJob: "
+ printJob.toString());
    }

    /*
     * (non-Javadoc)
     *
     * @see com.ibm.hats.runtime.IPrintJobManagerChangeListener#
     *      updatePrintJob(com.ibm.hats.runtime.PrintJobManager,
     *      com.ibm.hats.runtime.PrintJob)
     */
    public void updatePrintJob(PrintJobManager printJobManager,
                                             PrintJob printJob)
    {
        System.out.println("PrintExamplePlugin: updatePrintJob: "
                                            + printJob.toString());
```

```
}

/*
 * (non-Javadoc)
 *
 * @see com.ibm.hats.runtime.events.ServiceManagerListener#
 *      serviceChanged(com.ibm.hats.runtime.events.ServiceManagerEvent)
 */
public void serviceChanged(ServiceManagerEvent event)
{
    IService theService = event.getService();

    if (theService instanceof ISessionService)
    {
        ISessionService sessionService = (ISessionService) theService;

        int type = event.getType();
        if (type == ServiceManagerEvent.TYPE_SERVICE_CREATED)
        {
            sessionService.addSessionServiceListener(this);
        }
        else if (type == ServiceManagerEvent.TYPE_SERVICE_DESTROYED)
        {
            sessionService.removeSessionServiceListener(this);
        }
    }
}

/*
 * (non-Javadoc)
 *
 * @see com.ibm.hats.runtime.events.SessionStateListener#
 *      sessionStateChanged(com.ibm.hats.runtime.events.SessionStateEvent)
 */
public void sessionStateChanged(StateChangeEvent event)
{
    // If the session has change to operational then add a listener to
    // the PrintJobManager (if there is one and we haven't already done so).
    StateType state = event.getNewState();
    if (state == StateType.OPERATIONAL)
    {
        ISessionService sessionService = event.getSessionService();
        ClientSpecificInfo csi = sessionService.getRuntimeService()
             .getClientContainer().accessClient(sessionService.getClientId());

        if (csi != null)
        {
            String asiId = ApplicationSpecificInfo.
                                    createCompositeAsiId(sessionService.
                                    getApplicationId(),
                                                sessionService.getViewId());
            ApplicationSpecificInfo asi = csi.peekAll(asiId);
            if (asi != null)
            {
                PrintSpecificInfo psi = asi.getPrint();
                if (psi != null)
                {
                    PrintResourceHandler prh = psi.getResourceHandler();
                    if (prh != null)
                    {
                        PrintJobManager pjm = prh.getPrintJobManager();

                        // Add this PrintJobManager to our list of
                        // PrintJobManagers.
                        // This will return true if we haven't encountered
                        // this PrintJobManager before.
                        // In this case, add ourselves as a
```

```
                            // listener to this PrintJobManager.
                            if (printJobManagers.add(pjm))
                            {
                                pjm.addChangeListener(this);
                            }
                        }
                    }
                }
            }
        }

        /*
         * (non-Javadoc)
         *
         * @see com.ibm.hats.runtime.events.ISessionServiceListener#
         *      aboutToProcessCommand(com.ibm.hats.runtime.events.CommandEvent)
         */
        public void aboutToProcessCommand(CommandEvent event)
        {
        }

        /*
         * (non-Javadoc)
         *
         * @see com.ibm.hats.runtime.events.ISessionServiceListener#
         *      afterProcessCommand(com.ibm.hats.runtime.events.CommandEvent)
         */
        public void afterProcessCommand(CommandEvent event)
        {
        }
}
```

Creating a custom composite for use with the Show action

SWT Composites referenced in a Show Composite Action should implement the com.ibm.hats.rcp.transform.IRenderable interface. If the SWT Composite implements com.ibm.hats.rcp.transform.IRenderable, it will have access to the RcpContextAttributes that contains, among other things, the ISessionService for the session the composite is being shown in. A `sendContinue()` method on the ISessionService can be called to resume executing the HATS application.

To create a composite using the Visual Editor (see "Editing transformations" on page 19):

1. Select **File > New > Other > Java > Visual Class** to start the **New Java Visual Class** wizard.
2. Select the `src` folder for your plug-in project, and enter a name for the new composite class, for example, `MyComposite`
3. In the **Style** tree, select **SWT > Composite**.
4. Click **Add**, and select the com.ibm.hats.rcp.transform.IRenderable interface.
5. Click **Finish** to create the class.

To add a continue button to the composite using the Visual Editor (see "Editing transformations" on page 19):

1. From the **Palette** view, open the **SWT Controls** drawer and select **Button**.
2. Drag and drop the button on the composite. If prompted, accept the default name for the variable.
3. Right-click the selected button and select **Set Text**. Enter a caption for the button. For example, enter `OK` or `Submit`. Click **OK**.

4. Right-click the button again and select **Events > Add Events**. Select **Selection > widgetSelected** and click **Finish**.
5. Replace the temporary contents of the `widgetSelected(SelectionEvent)` method with the following:

```
RcpContextAttributes contextAttributes =
                        (RcpContextAttributes)getContextAttributes();
ISessionService sessionService = contextAttributes.getSessionService();
                                 sessionService.sendContinue();
```

Note: You might need to update the import section of your class. To do this, press CTRL+SHIFT+O or right-click in the source of your class and select **Source > Organize Imports**.

Chapter 7. Integrating business logic

Business logic is any Java code that is invoked as an action when an event occurs, such as a host screen being recognized or your HATS application being started. Business logic is specific to the application and is not provided as part of HATS. You can use business logic to extend your HATS application to integrate with other data sources, such as a database. For example, you can read the contents of a file or a database into HATS global variables and use the global variables to fill in a drop-down list or pop-up to be used in the application's GUI pages.

You can create business logic and add it to your project using the Create Business Logic wizard. To invoke this wizard, right-click in the **HATS Projects** tab of the HATS Toolkit, and click **New HATS > Business Logic**.

In the Create Business Logic wizard, specify the project you want to add the business logic to and supply the Java class name. The default package name is *projectName*.businessLogic, but you can change this in Studio Preferences. Optionally, you can supply a package name or click **Browse** to select an existing Java package. If you want your business logic to include methods for easy access to the project global variables, or to remove project global variables, select the **Create global variable helper methods** check box. Click **Finish** when you have provided the required information.

After you create your business logic class, you will want to link it to one or more screen or application events so it is executed when that event occurs. Edit each event (application event or screen customization) to which you want to add the link. On the Actions tab, click **Add**, select **Execute business logic**, then fill in the details for your business logic class. Refer to *HATS User's and Administrator's Guide* for information about editing both screen customization and events.

You can see the business logic files in the project by expanding the **Source** folder on the **HATS Project View** tab of the HATS Toolkit. Each package name or class name appears in the **Source** folder. Expand the package name folder to see the Java class name. Click the class name to edit the class. The **Source** folder can also include other Java files that have been imported into your HATS project.

If you use the Create Business Logic wizard to create business logic, the method that is invoked by the execute action is named **execute** by default. If you write your own class, the method must have the following attributes:

- Be marked public and static
- Have a return type of void
- Accept a com.ibm.hats.common.IBusinessLogicInformation object as the only parameter

The method must use this form, followed by your own business logic code:

```
public static void myMethod (IBusinessLogicInformation businesslogic)
```

The IBusinessLogicInformation object that is passed to your custom Java code enables you to access and use or modify various objects and settings of your HATS project. These include:

- The com.ibm.hats.runtime.IRequest class, which returns an object representing the request made to the HATS runtime and provides access to request parameters.
- The com.ibm.hats.runtime.IResponse class, which returns an object representing the response from the HATS runtime.
- The `getConnectionMap()` method, which returns a java.util.Map that contains the settings for the connection information that you provided for the application.
- The `getGlobalVariables()` method, which returns a java.util.Hashtable of global variables for this application instance. This table does not include shared global variables.
- The `getSharedGlobalVariables()` method, which returns a java.util.Hashtable of shared global variables for this application instance.
- Class properties, which provide default settings for objects such as components and widgets
- The com.ibm.hats.common.HostScreen object, which contains host screen information
- The java.util.Locale class of the client
- The com.ibm.hats.common.TextReplacementList values and settings
- The client session identifier string (returned by getter methods in the business logic template that the Create Business Logic wizard provides)
- The current screen orientation of bidirectional sessions
- The existence of the **Screen Reverse** button in the browser for bidirectional sessions

For more information about the classes made available to you, see the HATS API documentation in the HATS Knowledge Center at http://www.ibm.com/support/knowledgecenter/SSXKAY_9.5.0 for the IBusinessLogicInformation class. Since IBusinessLogicInformation extends the IBaseInfo class, several of these APIs are defined in the IBaseInfo class.

Incorporating Java code from other applications

You can incorporate Java code from other existing applications into your HATS projects in a variety of ways.

If you want to incorporate the source code (.java files) from your existing business logic so you can modify the code, you can import the .java files into the **Source** folder in your existing project. Click **File > Import > General > File System** to open the Import wizard. In the Import wizard, select the location of your source files in the **From directory** field. For RCP projects, select the **src** folder of your project in the destination **Into folder** entry field. When your source .java files are imported, they are automatically compiled and packaged into your HATS project. You can edit, set breakpoints, and debug your source files in the Rational SDP workbench.

You can also incorporate a Java archive (.jar) file with compiled Java business logic. The entire Java archive is added; you cannot select individual classes to add.

To import .jar files to RCP projects, use the following steps:

1. Create a directory (usually "lib" or "runtime") (optional).

2. Click **File > Import > General > File System** to open the Import wizard to import the .jar file, and select either the directory just created in step 1 or import to the root of the project if you did not create a directory as indicated in step 1.
3. Go to the **Navigator** view, and find the MANIFEST.MF file under the META-INF directory.
4. Double-click the file to open it in the manifest editor.
5. Go to **Runtime** tab, and under Classpath section, select **Add...**,
6. Browse to the imported .jar file, check **Update the build path**, and click **OK**.
7. Save the MANIFEST.MF file.

Using global variables in business logic

If your HATS application uses global variables to store information, you can use these global variables in your business logic.

There are two types of global variables: local and shared. A local global variable is one that is created within a particular RCP project and is only usable by that project. A shared global variable is one that is created in one or more RCP projects and can be used by all the applications running in the same user environment. There are also two lists of HATS global variables, one for local global variables and one for shared global variables. Two global variables with the same name can coexist if one is local and the other is shared.

When you create your business logic class, use the Create Business Logic wizard and select the **Create global variable helper methods** check box. This creates methods in your business logic for getting, setting, and removing local and shared global variables.

The following methods are created:

```
////////////////////////////////////////////////////////////////////////////
//  This sample is provided AS IS.
//  Permission to use, copy and modify this software for any purpose and
//  without fee is hereby granted. provided that the name of IBM not be used in
//  advertising or publicity pertaining to distribution of the software without
//  specific written permission.
//
//  IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SAMPLE, INCLUDING ALL
//  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL IBM
//  BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
//  DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
//  IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
//  OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SAMPLE.
////////////////////////////////////////////////////////////////////////////
/**
 * Example method that sets a named global variable
 * from the current session to a value
 * @param blInfo - IBusinessLogicInformation from current session
 * @param name - Name of the global variable
 * @param value - Value of the global variable
 */
public static void setGlobalVariable(IBusinessLogicInformation blInfo,
     String name,Object value)
{
  IGlobalVariable gv = blInfo.getGlobalVariable(name);
  if ( gv == null )
   {
    gv = new GlobalVariable(name,value);
   }
  else
   {
    gv.set(value);
   }
  blInfo.getGlobalVariables().put(name,gv);
}
```

```
/**
 * Example method that sets a named shared
 * global variable from the current session to a value
 * @param blInfo - IBusinessLogicInformation from current session
 * @param name - Name of the shared global variable
 * @param value - Value of the shared global variable
 */
public static void setSharedGlobalVariable(IBusinessLogicInformation
  blInfo,String name,Object value)
{
  IGlobalVariable gv = blInfo.getSharedGlobalVariable(name);
  if ( gv == null )
   {
    gv = new GlobalVariable(name,value);
   }
  else
   {
    gv.set(value);
   }
  blInfo.getSharedGlobalVariables().put(name,gv);
}

/**
 * Example method that removes a named global variable from the current session
 * @param blInfo - IBusinessLogicInformation from current session
 * @param name - Name of the global variable
*/
public static void removeGlobalVariable(IBusinessLogicInformation blInfo, String  name)
{
  IGlobalVariable gv = blInfo.getGlobalVariable(name);
  if ( gv != null )
   {
    blInfo.getGlobalVariables().remove(name);
    gv.clear();
    gv = null;
   }
}
/**
 * Example method that removes a named shared global variable from the current session
 * @param blInfo - IBusinessLogicInformation from current session
 * @param name - Name of the shared global variable
*/
public static void removeSharedGlobalVariable(IBusinessLogicInformation blInfo, String name)
{
  IGlobalVariable gv = blInfo.getSharedGlobalVariable(name);
   if ( gv != null )
    {
     blInfo.getSharedGlobalVariables().remove(name);
     gv.clear();
     gv = null;
    }
}
/**
 * Example method that retrieves a named global variable from the current session
 * @param blInfo - IBusinessLogicInformation from current session
 * @param name - Name of the global variable
 * @return - an instance of the global variable, or null if not found.
 */
public static IGlobalVariable getGlobalVariable(IBusinessLogicInformation
  blInfo,String name)
{
  IGlobalVariable gv = blInfo.getGlobalVariable(name);
  return gv;
}

/**
 * Example method that retrieves a named shared
 * global variable from the current session
 * @param blInfo - IBusinessLogicInformation from current session
 * @param name - Name of the shared global variable
 * @return - an instance of the global variable, or null if not found.
 */
public static IGlobalVariable getSharedGlobalVariable(IBusinessLogicInformation
  blInfo,String name)
{
  IGlobalVariable gv = blInfo.getSharedGlobalVariable(name);
  return gv;
}
```

Elsewhere in your code, when you need the value of a local global variable, you can call this method:

```
GlobalVariable gv1 = getGlobalVariable(blInfo,"varname");
```

To get the value of a shared global variable, use the following method:

```
GlobalVariable gv1 = getSharedGlobalVariable(blInfo,"varname");
```

Business logic examples

This section contains examples of using business logic. Each works with global variables. Each example uses one or more of the global variable helper methods previously described, and the classes should include those methods. They are omitted in these examples to make it easier to view the example code.

Example: Date conversion

This example converts a date from mm/dd/yy format to month, day, year format. For example, the example converts 6/12/2004 into June 12, 2004. The example assumes that the global variable *theDate* has been set before the business logic is called. Note how the example uses the following method to obtain the value of the input variable:

```
IGlobalVariable inputDate = getGlobalVariable(blInfo, "theDate");
```

After using standard Java functions to manipulate the string to represent the date in the desired format, the example uses the following method to put the new string into the same global variable:

```
setGlobalVariable(blInfo, "theDate", formattedDate);
```

```
////////////////////////////////////////////////////////////////////////////
//  This sample is provided AS IS.
//  Permission to use, copy and modify this software for any purpose and
//  without fee is hereby granted. provided that the name of IBM not be used in
//  advertising or publicity pertaining to distribution of the software without
//  specific written permission.
//
//  IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SAMPLE, INCLUDING ALL
//  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL IBM
//  BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
//  DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
//  IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
//  OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SAMPLE.
////////////////////////////////////////////////////////////////////////////
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

import com.ibm.hats.common.IBusinessLogicInformation;
import com.ibm.hats.common.GlobalVariable;
import com.ibm.hats.common.IGlobalVariable;

public class CustomDateFormatter
{
   public static void execute(IBusinessLogicInformation blInfo)
   {
    IGlobalVariable inputDate = getGlobalVariable(blInfo, "theDate");
    SimpleDateFormat inputFormat = new SimpleDateFormat("MM/dd/yyyy");
    SimpleDateFormat outputFormat = new SimpleDateFormat("MMMM dd, yyyy");

    try
     {
      Date tempDate = inputFormat.parse(inputDate.getString().trim());
      String formattedDate = outputFormat.format(tempDate);
      setGlobalVariable(blInfo, "theDate", formattedDate);
     }
      catch (ParseException ex)
       {
```

```
        ex.printStackTrace();
       }
   }
}
```

Example: Adding values that are contained in an indexed global variable

This example adds the values that are contained in an indexed global variable and stores the sum in another, non-indexed global variable. It assumes that you have stored strings representing numbers in the indexed global variable *subtotals*.

The previous example included the names of the input and output global variables (*theDate*) on the set calls. This example sets the names of the input and output variables into local string variables and uses those strings on calls to get and set the global variable values. Because the name of the global variable is being passed as a variable, it is not put into quotes:

```
setGlobalVariable(blInfo,gvOutputName, new Float(myTotal));
////////////////////////////////////////////////////////////////////////////
//  This sample is provided AS IS.
//  Permission to use, copy and modify this software for any purpose and
//  without fee is hereby granted. provided that the name of IBM not be used in
//  advertising or publicity pertaining to distribution of the software without
//  specific written permission.
//
//  IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SAMPLE, INCLUDING ALL
//  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL IBM
//  BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
//  DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
//  IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
//  OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SAMPLE.
////////////////////////////////////////////////////////////////////////////
import com.ibm.hats.common.IBusinessLogicInformation;
import com.ibm.hats.common.GlobalVariable;
import com.ibm.hats.common.IGlobalVariable;

  public static void execute(IBusinessLogicInformation blInfo)
    {
     // Name of indexed global variable to be read in
        String gvInputName = "subtotals";
        // Name of global variable to be calculated and saved
        String gvOutputName = "total";

        // The indexed global variable where each index is a subtotal to be summed
        GlobalVariable gvSubtotals =
     ((GlobalVariable)getGlobalVariable(blInfo, gvInputName));

        float myTotal = 0;

        // Calculate the total by adding all subtotals
        for (int i = 0; i < gvSubtotals.size(); i++)
        {
         myTotal = myTotal + Float.valueOf(gvSubtotals.getString(i)).floatValue();
        }

        // Save the total as a non-indexed local variable
        setGlobalVariable(blInfo,gvOutputName, new Float(myTotal));
    }
```

Example: Reading a list of strings from a file into an indexed global variable

This example reads a file from the file system and stores the strings in the file into an indexed global variable. You can use a technique like this to read a file that contains, for example, a list of your company's locations. After storing the strings in a global variable, you can use the global variable to populate a drop-down list or other widget to enable users to select from a list of values. You can create a global rule to use this widget wherever an appropriate input field occurs. To make

sure that the global variable is available as soon as the application is started, add the execute action for this business logic class to the Start event.

Note: If your text file has carriage returns and line feeds between lines, you might need to use "\r\n" as the second argument of the StringTokenizer constructor call in the following example.

```
////////////////////////////////////////////////////////////////////////////
//  This sample is provided AS IS.
//  Permission to use, copy and modify this software for any purpose and
//  without fee is hereby granted. provided that the name of IBM not be used in
//  advertising or publicity pertaining to distribution of the software without
//  specific written permission.
//
//  IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SAMPLE, INCLUDING ALL
//  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL IBM
//  BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
//  DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
//  IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
//  OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SAMPLE.
////////////////////////////////////////////////////////////////////////////
import com.ibm.ejs.container.util.ByteArray;
import com.ibm.hats.common.IBusinessLogicInformation;
import com.ibm.hats.common.GlobalVariable;
import com.ibm.hats.common.IGlobalVariable;

public class ReadNamesFromFile
{

  public static void execute(IBusinessLogicInformation blInfo)
  {
  // Name of indexed global variable to be saved
  String gvOutputName = "namesFromFile";

  // The file containing a list of information (in this case, it contains names)
  java.io.File myFileOfNames = new java.io.File("C:" + java.io.File.separator
       + "temp" + java.io.File.separator + "names.txt");

     try
     {
   // First, read the contents of the file

   java.io.FileInputStream fis = new java.io.FileInputStream(myFileOfNames);

   int buffersize = (int)myFileOfNames.length();
   byte[] contents = new byte[buffersize];;

   long n = fis.read(contents, 0, buffersize);

   fis.close();

   String namesFromFile = new String(contents);

   // Next, create an indexed global variable from the file contents

   java.util.StringTokenizer stok =
      new java.util.StringTokenizer(namesFromFile, "\n", false);

   int count = stok.countTokens();
   String[] names = new String[count];
   for (int i = 0; i < count; i++)
   {
    names[i] = stok.nextToken();
   }

   IGlobalVariable gv = new GlobalVariable(gvOutputName, names);
   blInfo.getGlobalVariables().put(gvOutputName,gv);
     }
     catch (java.io.FileNotFoundException fnfe)
     {
      fnfe.printStackTrace();
     }
     catch (java.io.IOException ioe)
     {
      ioe.printStackTrace();
     }
  }
}
```

Using custom screen recognition

You can use business logic to perform custom screen recognition. HATS Toolkit provides many options for recognizing screens within a screen customization, including the number of fields on a screen, the presence or absence of strings, and the use of global variables. These options are described in *HATS User's and Administrator's Guide*. You might find, however, that you want to recognize a screen in a way that you cannot configure using the options in the screen customization editor. In that case, you can add your own custom screen recognition logic.

Note: The information in this section can be used for screen recognition within macros as well as within screen customizations.

If you want to create custom screen recognition logic using HATS global variables, see "Custom screen recognition using global variables" on page 62.

If you have already created custom screen recognition logic by extending the ECLCustomRecoListener class, you can use this logic within HATS. If you are creating new custom logic, follow these steps:

1. Open the Java perspective.
2. Click **File > New > Class**.
3. Browse to the Source directory of your HATS project.
4. Enter the names of your package and class.
5. For the superclass, click **Browse** and locate com.ibm.hats.common.customlogic.AbstractCustomScreenRecoListener.
6. Select the check box for **Inherited abstract methods**. Click **Finish**. This imports the code skeleton into the project you specified.
7. Add your logic to the `isRecognized` method. Make sure that it returns a boolean value.

   ```
   public boolean isRecognized(String arg0, IBusinessLogicInformation
   arg1, ECLPS arg2, ECLScreenDesc arg3)
   ```

   Refer to the HATS API documentation at the HATS Knowledge Center for a description of this method.
8. After creating your method, you must update the screen recognition to invoke your method. From the HATS Projects view, expand your project and the **Screen Customizations** folder. Double-click the name of the screen customization to which you want to add your custom logic. Click the **Source** tab to open the Source view of the screen customization.
9. Within the Source view, you will see a block that begins and ends with the <description> and </description> tags. This block contains the information that is used to recognize screens. Add a line within this block to invoke your custom logic:

   ```
   <customreco id="customer.class.package.MyReco::settings"
   invertmatch="false" optional="false"/>
   ```

   where customer.class.package.MyReco is your package and class name. If you want to pass any settings into your class, add them after the class name, separated by two colons. Settings are optional, and your class must parse whatever values are passed in. If you do not need settings, omit the two colons.

   Consider where within the description block you want to place the <customreco> tag. If you want your custom logic invoked only if all the other criteria match, place the <customreco> tag at the end of the block,

immediately before the </description> tag. If your screen customization compares a screen region to a value, the description block will contain a smaller block, beginning and ending with the <block> and </block> tags, to define the value to which the screen region is compared. Be sure not to place your customreco tag inside this block.

Following is an example section of a description block. Note the <customreco> tag just before the </description> tag, and not between the <block and </block> tags.

```
<description>
<oia invertmatch="false" optional="false" status="NOTINHIBITED"/>
<numfields invertmatch="false" number="61" optional="false"/>
<numinputfields invertmatch="false" number="16" optional="false"/>
<block casesense="false" col="2" ecol="14" erow="21"
   invertmatch="false" optional="false" row="20">
<string value="USERID ==="/>
<string value="PASSWORD ==="/>
</block>
<cursor col="16" invertmatch="false" optional="false" row="20"/>
<customreco id="customer.class.package.MyReco::settings"
   invertmatch="false" optional="false"/>
</description>
```

10. To rebuild your HATS project, click **Project > Clean** on the toolbar.
11. For RCP, run the application in your local test environment to test your project. Refer to *HATS Getting Started* for more information.

Example of custom screen recognition

Following is an example of business logic that performs custom screen recognition. This business logic class takes a list of code page numbers, separated by blanks, as its settings, and recognizes the screen if its code page matches one of those listed in the settings. The tag syntax is:

```
<customreco id="company.project.customlogic.CodePageValidate::[settings]"
optional="false" invertmatch="false" />
```

For example, you can insert the following tag into a description block:

```
<customreco id="company.project.customlogic.CodePageValidate::037 434 1138"
optional="false" invertmatch="false" />
```

In this case the screen will be recognized if its code page is 037, 434, or 1138.

```
////////////////////////////////////////////////////////////////////////////
//  This sample is provided AS IS.
//  Permission to use, copy and modify this software for any purpose and
//  without fee is hereby granted. provided that the name of IBM not be used in
//  advertising or publicity pertaining to distribution of the software without
//  specific written permission.
//
//  IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SAMPLE, INCLUDING ALL
//  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL IBM
//  BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
//  DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER
//  IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING
//  OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SAMPLE.
////////////////////////////////////////////////////////////////////////////
package company.project.customlogic;

import java.util.StringTokenizer;
import java.lang.Integer;
import com.ibm.eNetwork.ECL.ECLPS;
import com.ibm.eNetwork.ECL.ECLScreenDesc;
import com.ibm.hats.common.IBusinessLogicInformation;
import com.ibm.hats.common.HostScreen;
import com.ibm.hats.common.customlogic.AbstractCustomScreenRecoListener;

public class CodePageValidate extends AbstractCustomScreenRecoListener {
```

```
/**
 * @see com.ibm.hats.common.customlogic.AbstractCustomScreenRecoListener
 * #isRecognized(java.lang.String, com.ibm.hats.common.IBusinessLogicInformation,
 *       com.ibm.eNetwork.ECL.ECLPS,  com.ibm.eNetwork.ECL.ECLScreenDesc)
  */
 public boolean isRecognized(
  String settings,
  IBusinessLogicInformation bli,
  ECLPS ps,
  ECLScreenDesc screenDescription) {
  HostScreen hs=bli.getHostScreen();
  int int_codepage=hs.GetCodePage();
  if(settings!=null)
  {
   StringTokenizer tokenizer = new StringTokenizer(settings);
   while( tokenizer.hasMoreTokens() )
   {
    int int_token= Integer.valueOf(tokenizer.nextToken()).intValue();
    if ( int_token==int_codepage )
    {
      return true;
    }
   }
  }
  return false;
 }

}
```

Custom screen recognition using global variables

HATS Toolkit provides some screen recognition options using global variables, including these functions:

- Verify that a global variable exists
- Verify that a global variable does not exist
- Verify the integer or string value of a global variable

Refer to *HATS User's and Administrator's Guide* for information about these options. If you want to perform screen recognition that is based on HATS global variables and the options in the Global Variable Logic panel do not meet your requirements, you can add your own logic based on the values or existence of one or more global variables. This approach does not require you to create a Java class; instead, it uses the GlobalVariableScreenReco class, which is provided by HATS, and you can specify comparisons to be made as settings on the <customreco> tag. The format is one of the following:

- ```
  <customreco
  id="com.ibm.hats.common.customlogic.GlobalVariableScreenReco::
  {variable(name,option,resource, index)}COMPARE{type(name,option,resource,
                                                      index)}"
    invertmatch="false" optional="false"/>
  ```
- ```
  <customreco
  id="com.ibm.hats.common.customlogic.GlobalVariableScreenReco::
  {variable(name,option,resource, index)}COMPARE{type(value)}"
  invertmatch="false" optional="false"/>
  ```

Braces {} are used to contain each of the two items that are being compared. The first item is a HATS global variable, whose name is specified in *name*. You can use `option` to specify that you want to use the variable's value, length, or existence in your comparison. The `resource` and `index` settings are optional. Use `resource` to indicate whether the global variable is local (which is the default) or shared. Use `index` to indicate which value to use from an indexed global variable.

The second item can be one of the following:

- Another HATS global variable, with similar options, in which case the first format is used
- A fixed value, in which case the second format is used

The valid values for the settings are shown in Table 17. For the COMPARE setting, the only valid values for comparing strings are EQUAL and NOTEQUAL.

Table 17. Valid values for settings

Setting	Valid values
type	• variable • integer • boolean • string
COMPARE	• EQUAL • NOTEQUAL • GREATERTHAN • GREATERTHANOREQUAL • LESS THAN • LESSTHANOREQUAL
options	• exists (boolean) • value (string/integer/boolean) • length (integer) • object (object)
resource	• local • shared
index	Any positive integer or 0

The following example compares the values of two local global variables:

```
<customreco id="com.ibm.hats.common.customlogic.GlobalVariableScreenReco::
  {variable(name=gv1,option=value,resource=local)}EQUAL
  {variable(name=gv2,option=value,resource=local)}"
  invertmatch="false" optional="false"/>
```

This expression evaluates to true if the values of *gv1* and *gv2* are the same.

Now consider the length option. For a non-indexed global variable, length is the length of the value of the variable. For an indexed global variable, if you specify an index, length is the length of that index of the global variable; if you do not specify an index, length is the number of indexed entries in the global variable.

```
<customreco id="com.ibm.hats.common.customlogic.GlobalVariableScreenReco::
  {variable(name=gv1,option=length,resource=shared)}LESSTHANOREQUAL
  {variable(name=gv2,option=length,index=4)}" invertmatch="false" optional="false"/>
```

This expression compares the length of *gv1* to the length of the fourth index of *gv2*. It evaluates to true if the length of *gv1* is less than or equal to the length of the fourth index of *gv2*. You can use LESSTHANOREQUAL because length returns an integer value.

The use of resource=shared on *gv1* in this example indicates that *gv1* is a shared global variable. The other option is resource=local, which is the default, and means that the global variable is not shared with other applications.

You do not have to compare one global variable with another. You can compare the length of a global variable with a fixed integer. You can compare the value of a global variable with another string. For example:

```
<customreco id="com.ibm.hats.common.customlogic.GlobalVariableScreenReco::
  {variable(name=gv1,option=value)}EQUAL{string(value=mystring)}"
  invertmatch="false" optional="false"/>
```

This expression compares the value of *gv1* with the string that is contained in mystring. The string can be a fixed string, the value of a variable, or a value that is returned from a method call. In general, you do not need to use custom logic to compare the length or value of a global variable to a fixed value; you can add these comparisons using the Global Variable Logic panel.

Chapter 8. Creating custom components and widgets

HATS provides a set of host components that recognize elements of the host screen and widgets that render the recognized elements. The components and widgets have settings that you can modify if the default settings do not recognize components or render widgets as you want them. If the components, widgets, and the settings that are provided by HATS do not meet your needs, you can create your own custom components or widgets or modify existing host components or widgets. You might want to create your own host component in order to recognize elements of your host screen that the HATS components do not recognize. You might want to create your own widget in order to change the way elements are presented on the page. The following sections describe how to create custom host components and widgets. For further information, see Appendix A, "HATS Toolkit files," on page 81.

Note: If you are using a bidirectional code page, you can control the direction of widgets and other presentation aspects. See Chapter 9, "Using the HATS bidirectional API," on page 75.

Components and widgets properties for RCP applications

In RCP projects, transformations are SWT (Standard Widget Toolkit) composites. RCP transformations have a .java file extension. Transformations can be found in the HATS Projects view under **Rich Client Content > Transformations**. Transformations are made up of ComponentRendering composites (a specialized composite or panel that transforms a given region using the specified component, widget, and settings). The following code example illustrates the concept:

```
ComponentRendering rendering1 = new ComponentRendering(this, 0);
    rendering1.setComponent("<fully qualified component class name>");
    rendering1.setWidget("<fully qualified widget class name>");
    rendering1.setRegion(new BlockScreenRegion(start_row, start_col, end_row, end_col));
    rendering1.setComponentSettings(new StringableProperties("<component settings>"));
    rendering1.setWidgetSettings(new StringableProperties("<widget settings>"));
    rendering1.setHostScreen(getHostScreen());
    rendering1.setTransformInfo(getTransformInfo());
    rendering1.setAutoRender(false);
    rendering1.setTextReplacement("<text replacement string>");
    rendering1.render();
```

In the above example, *component settings* means a string of component settings and *widget settings* means a string of widget settings.

These composites are usually constructed and rendered within the `render()` method of the transformation.

SWT widgets must implement the com.ibm.hats.rcp.transform.renderers.SwtRenderer interface for RCP projects, and SWT widgets have a `drawSwt` method.

The following SWT widget code sample shows the constructor and `drawSwt` method:

```
public MyCustomWidget extends Widget implements SwtRenderer {

    public MyCustomWidget (ComponentElement[] componentElements, Properties settings) {
        super(componentElements, settings);
    }
```

```
    public Control[] drawSwt(Composite parent) {
        SwtElementFactory elementFactory = SwtElementFactory.newInstance(getContextAttributes(),
        getSettings(), parent.getDisplay());

        // Add code here to generate SWT Control objects here

         return controls;  // returns controls added to the specified parent
    }
}
```

Creating a custom host component

HATS provides a Create Component wizard to help you create custom components. You can start the wizard in several ways:

- From the **File > New > HATS Component** menu in HATS Toolkit
- From the **HATS > New > Component** menu in HATS Toolkit
- From the context (right-click) menu of a HATS project, select **New HATS > Component**

There are two panels of the Create Component wizard. On the first panel, you provide the name of the project, the name of the new component, and the name of the Java package for the component. Optionally, you can select a check box to include stub methods that allow you to define a GUI panel for configuring the settings used by the new component (see "HATS Toolkit support for custom component and widget settings" on page 72 for more information). On the second panel, you enter the name you want displayed for the new component in the HATS Toolkit and select the widgets to associate with the component. Widget association is not necessary to complete the wizard. You can define the association of components and widgets later. Refer to "Registering your component or widget" on page 70 for more information about associating components and widgets.

The following sections explain the required elements of a custom component that the wizard provides:

- Extends the host component abstract class, `com.ibm.hats.transform.components.Component`.

  If one of the HATS host components is very similar to what you need, it will be easier to extend that component. See "Extending component classes" on page 68 for more information.
- Adds the constructor method. This method, named for your component, must accept a com.ibm.hats.common.HostScreen object. For example:

  ```
  public MyComponent(HostScreen hostScreen) {
    super(hostScreen);
  }
  ```

  The constructor should initialize parameters that the `recognize()` method will require, based on the host screen object.
- Adds the `recognize()` method.

  ```
  public ComponentElement[] recognize(BlockScreenRegion region,
                                      Properties settings)
  ```

  The `recognize()` method has a different implementation in each host component class. It accepts the region and settings passed to it and returns an array of component element objects. You should implement this method to implement your own pattern recognition logic.

  The `recognize()` method must return an array of ComponentElement objects, as defined in com.ibm.hats.transform.elements.ComponentElement. Each HATS

component returns a slightly different set of elements that extend ComponentElement. For example, the SelectionListComponent returns an array of SelectionComponentElement objects. This array of component elements is passed to the specified widget, so be sure to return an array of elements that can be accepted by the widget you want to use.

For a description of the arguments of this method, refer to the HATS API References (Javadoc) for the `recognize()` method of the Component class. See "Using the API documentation (Javadoc)" on page 2.

- Adds the source code for the component into the **Source** folder of your project
- Compiles the new component .java file, if you have **Build Automatically** checked in the Rational SDP workbench preferences (**Window > Preferences > General > Workspace** or **Project > Build Automatically**). If the component is not compiled into a .class file, it is not available for use in the HATS Toolkit.
- Registers the new component in the `ComponentWidget.xml` file. See "Registering your component or widget" on page 70 for more information about registering components.

If you selected the check box to include HATS Toolkit graphical user interface support methods, enabling you to modify the settings of the component, the Create Component wizard adds the following methods:

- Method to return the number of pages in the property settings:

```
public int getPropertyPageCount() {
  return (1);
}
```

- Method to return the settings that can be customized:

```
public Vector getCustomProperties(int iPageNumber, Properties properties,
   ResourceBundle bundle) {
  return (null);
}
```

- Method to return the default values of the settings that can be customized:

```
public Properties getDefaultValues(int iPageNumber) {
  return (super.getDefaultValues(iPageNumber));
 }
```

See "HATS Toolkit support for custom component and widget settings" on page 72 for more information about the methods necessary to support your custom component.

Note: If you want your component to work properly within Default Rendering, you must set the consumed region (that is, the area of the host screen that has been processed) on each component element that your component returns, before returning the component element. This tells the Default Rendering that this region of the screen has been consumed, or processed, by a host component and should not be processed again. To set the consumed region, use this method:

```
public void setConsumedRegion(BlockScreenRegion region)
```

Refer to the HATS API References (Javadoc) for the ComponentElement class for more information. See "Using the API documentation (Javadoc)" on page 2.

Extending component classes

HATS provides a number of host component classes. You can extend any of the host component classes that are found in the `ComponentWidget.xml` file by replacing the statement `public class MyCustomComponent extends Component` in the created .java file for the new component with the class name of an existing component. For example:

```
public class MyCustomComponent
             extends com.ibm.hats.transform.components.CommandLineComponent
```

Note: Bidirectional components are stored in the com.ibm.hats.transform.components.BIDI package. The names of bidirectional classes for components are the same as regular components, but they are followed by "BIDI"; for example, com.ibm.hats.transform.components.BIDI.CommandLineComponentBIDI.

Each HATS component performs recognition of elements of the host screen in the `recognize()` method. To extend a host component and accomplish the specific recognition task you need, you can use either of these approaches:

- Extend one of the component classes that is provided by HATS and override the `recognize()` method of the component. Somewhere in your `recognize()` method you should add a call like `super.recognize(region, settings);` to invoke the `recognize()` method of the class you extended. You can modify the process by changing the settings before calling the superclass, or by manipulating the output returned by the superclass.
- Extend one of the component classes that is provided by HATS and override the `recognize()` method of the component. Instead of using the `recognize()` method of the superclass, invoke the `recognize()` method of one of the other component classes. This approach will be useful if you want to recognize a complex host component that combines aspects of more than one of the HATS components.

The Create Component wizard generates a recognize() method that returns null, which indicates that the host screen region is not recognized by the new component. To change the custom component to act as the HATS component it is extended from, whose elements contain all of the correct ComponentElements, remove the "return null" from the .java file and change the code in the component code. For example:

```
public ComponentElement[] recognize(LinearScreenRegion region, Properties settings) {
ComponentElement [] elements = super.recognize(region, settings);
return elements;
}
```

To edit the `ComponentWidget.xml` file, click the **Navigator** tab of the HATS Toolkit. The `ComponentWidget.xml` file is shown at the bottom of the **Navigator** view of your project. See "Registering your component or widget" on page 70 for more information about the `ComponentWidget.xml` file.

Creating a custom widget

HATS provides a Create Widget wizard to help you create custom widgets. You can start the wizard in several ways:

- From the **File > New > HATS Widget** menu in HATS Toolkit
- From the **HATS > New > Widget** menu in HATS Toolkit
- From the context (right click) menu of a HATS project, select **New HATS > Widget**

There are two panels in the Create Widget wizard. On the first panel, you provide the name of the project, the name of the new widget, and the name of the Java package for the widget. Optionally, you can select a check box to include stub methods that allow you to define a GUI panel for configuring the settings used by the new widget (see "HATS Toolkit support for custom component and widget settings" on page 72 for more information). On the second panel, enter the name you want displayed for the new widget in the HATS Toolkit and select the components to associate with the widget.

The wizard provides the following required elements of a custom widget:

- Extends the widget abstract class and implements the SwtRenderer interface: `public class MyCustomWidget extends Widget implements SwtRenderer`.

  See "Extending widget classes" on page 70 for more information.
- Adds the constructor method:

```
public MyCustomWidget(ComponentElement[] arg0, Properties arg1) {
  super(arg0, arg1);
}
```

- Adds the following method to generate SWT for RCP project output.

```
public Control[] drawSwt(Composite parent) {
        SwtElementFactory elementFactory =
                 SwtElementFactory.newInstance(contextAttributes,
                                  settings, parent.getDisplay());

        // Construct controls using SwtElementFactory instance

        return new Control[0];
    }
```

  The SwtElementFactory should be used to construct SWT controls since it handles associating new controls with the underlying data model and applying the styles of the associated template.
- Adds the source code for the widget into the **Source** folder of your project
- Compiles the new widget .java file, if you have **Build Automatically** selected in the Rational SDP workbench preferences (**Window > Preferences > General > Workspace**) or the Project menu. If the widget is not compiled into a .class file, it is not available for use in the HATS Toolkit.
- Registers the new widget in the `ComponentWidget.xml` file. See "Registering your component or widget" on page 70 for more information about registering widgets.

If you selected the check box to include HATS Toolkit graphical user interface support methods, enabling you to modify the settings of the widget, the Create Widget wizard adds the following methods:

- Method to return the number of pages in the property settings:

```
public int getPropertyPageCount() {
  return (1);
}
```

- Method to return the settings that can be customized:

```
public Vector getCustomProperties(int iPageNumber, Properties properties,
   ResourceBundle bundle) {
  return (null);
}
```

- Method to return the default values of the settings that can be customized:

```
public Properties getDefaultValues(int iPageNumber) {
  return (super.getDefaultValues(iPageNumber));
 }
```

See “HATS Toolkit support for custom component and widget settings” on page 72 for more information about the methods necessary to support your custom widget.

Extending widget classes

HATS provides a number of widget classes. You can extend any of the widget classes found in the `ComponentWidget.xml` file by replacing the

```
public class MyCustomWidget extends Widget implements SwtRenderer
```

in the created .java file for the new widget with the class name of an existing widget, such as

```
public class MyCustomWidget extends
  com.ibm.hats.rcp.transform.widgets.SwtFieldWidget
```

Note: Bidirectional widgets are stored in the com.ibm.hats.transform.widgets.BIDI package. The names of bidirectional classes for widgets are the same as regular widgets, but they are followed by “BIDI”; for example,

```
public class newBIDIField extends
  com.ibm.hats.rcp.transform.widgets.BIDI.SwtFieldWidgetBIDI implements SwtRenderer
```

If you want to modify an existing widget, you must extend one of the existing widget classes and override its `drawSwt` method. Refer to the HATS API References (Javadoc) for details about widget interfaces and methods. See “Using the API documentation (Javadoc)” on page 2.

Widgets and global rules

Widgets that present input fields should check whether the input field has already been processed by a HATS global rule. When a host screen is received, HATS searches it for host components that match global rules that are defined for that HATS application. When your widget checks whether the input field has already been processed by a HATS global rule, the call returns null if the input field has not been processed. If the input field has already been processed according to a global rule, the call returns the transformation fragment to which the input field has been transformed by the global rule. Your widget should output the fragment rather than processing the component element.

```
Control control = RcpRenderingRulesEngine.processMatchingElement(parent, fce,
                  contextAttributes);
                     if (control == null) {
                          ...
                      }
```

Add the above example to the drawSwt() method for the widget.

Registering your component or widget

Registering your custom components and widgets in the `ComponentWidget.xml` file makes them available for use in the HATS Toolkit, such as in the Insert Host Component wizard.

Host components must map to specific widgets. Custom host components can map to any existing widget or to a custom widget. The Create a custom component or widget wizards register your custom components and widgets in the `ComponentWidget.xml` file, and associates components and widgets. When using the

wizards, if you did not associate your custom component or widget, you need to edit the ComponentWidget.xml file and add the associations. To edit the ComponentWidget.xml file, click the **Navigator** tab of the HATS Toolkit. The ComponentWidget.xml file is shown at the bottom of the **Navigator** view of your project.

Note: If you decide to delete a custom component or widget after it has been registered, simply deleting the source code for the component or widget from the **Source** folder of your project is not enough to completely remove it. It is still referenced in the registry and there is no programmatic way to remove it. You should remove it from the registry by editing the ComponentWidget.xml file and deleting the references to the component or widget.

Following is an example of the ComponentWidget.xml file that shows the HATS-supplied Field Table component and one of the associated widgets, the vertical bar graph widget.

```
<ComponentWidgetList>
  <components>
    <component className="com.ibm.hats.transform.components.FieldTableComponent"
       displayName="Field table" image="table.gif">
      <associatedWidgets>
        <widget className="com.ibm.hats.rcp.transform.widgets.SwtVerticalBarGraphWidget"/>
      </associatedWidgets>
    </component>
  </components>


  <widgets>
    <widget className="com.ibm.hats.rcp.transform.widgets.SwtVerticalBarGraphWidget"
       displayName="Vertical graph" image="verticalBarGraph.gif" />
  </widgets>
</ComponentWidgetList>
```

As you can see, there are two sections to this file: components and widgets.

The components section contains the list of all registered components. To register a custom component and make it available to the HATS Toolkit, add a <component> tag and the associated <widget> tags to the ComponentWidget.xml file. You must supply a className, displayName, and the associated widgets.

className
Identifies the Java class that contains the code to recognize elements of the host screen. The class name is usually in the form com.*myCompany.myOrg.ClassName*.

displayName
Identifies the name by which your custom component is known and how it appears in the list of components in the HATS Toolkit. This name must be unique among the registered components. The form of the displayName for a custom component is simply a string. Spaces are allowed in the displayName.

image The image attribute identifies the image to use for your component when it appears in the HATS Toolkit.

widget
Identifies the widgets that are associated with this component. There must be a separate <widget> tag for each associated widget. All of the <widget> tags for the component must be defined within the <associatedWidgets> tag and its </associatedWidgets> ending tag. The <widget> tag within the

<associatedWidgets> tag contains only the className attribute, which identifies the Java class that contains the code to link the widget to the component. The class name is usually in the form com.*myCompany.myOrg.ClassName*.

The widgets section contains the list of all registered widgets. To register a widget, link it to a component, make it available for use in the HATS Toolkit, and add a <widget> tag to the `ComponentWidget.xml` file. You must supply a className and a displayName.

className
Identifies the Java class that contains the code to render the widget. The class name is usually in the form com.*myCompany.myOrg.ClassName*.

displayName
Identifies the name by which your custom widget is known and how it appears in the list of widgets in the HATS Toolkit. This name must be unique among the registered widgets. The form of the displayName for a custom widget is simply a string. Spaces are not allowed in the displayName. However, you can use an underscore (_) in place of a space.

HATS Toolkit support for custom component and widget settings

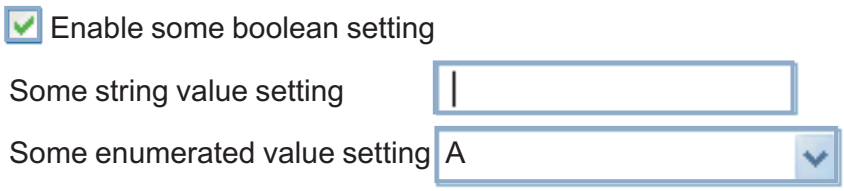
You can provide GUI support for modifying the settings of your custom component and widget. This is useful if other developers will be using your custom component or widget or you want to easily test different combinations of settings using the preview features available in the HATS Toolkit. The base component and widget classes implement the ICustomPropertySupplier interface. This interface allows a component or widget to contribute setting information to the HATS Toolkit. This information is used to render a panel by which the settings of the component or widget can be modified. Not all settings need to be exposed in the GUI.

The `getCustomProperties()` method returns a vector of **HCustomProperty** customizable property objects. Each **HCustomProperty** object represents a setting of the component or widget. The HATS Toolkit renders each **HCustomProperty** objects based on its type. For example, an object of type HCustomProperty.TYPE_BOOLEAN is rendered as a GUI checkbox.

The following sample code demonstrates how a widget can provide GUI support for three of its settings (mySetting1, mySetting2, and mySetting3):

```
// Returns the number of settings panels (property pages) to be contributed
//by this widget.  The returned value must be greater than or equal to 1 if
//custom properties will be supplied via the getCustomProperties() method.
public int getPropertyPageCount() {
         return 1;
}

// Returns a Vector (list) of custom properties to be displayed in the GUI
//panel for this component or widget.
public Vector getCustomProperties(int iPageNumber, Properties properties,
         ResourceBundle bundle) {
         Vector props = new Vector();

// Constructs a boolean property that will be rendered as a checkbox
HCustomProperty prop1 = HCustomProperty.new_Boolean("mySetting1",
                 "Enable some boolean setting", false, null, null);
props.add(prop1);
```

```
// Constructs a string property that will be rendered as a text field
HCustomProperty prop2 = HCustomProperty.new_String("mySetting2",
                "Some string value setting", false, null,
                 null, null, null);
props.add(prop2);

// Constructs an enumeration property that will be rendered as a drop-down
HCustomProperty prop3 = HCustomProperty.new_Enumeration("mySetting3",
                 "Some enumerated value setting", false,
                 new String[] { "A", "B", "C" }, new String[]
                 { "Option A", "Option B", "Option C" }, null, null, null);
props.add(prop3);

return props;     }
```



The values supplied by the user of the custom component or widget will be available in the *componentSettings* **Properties** object passed into the `recognize()` method of the component or the *widgetSettings***Properties** object passed into the constructor of the widget. The `getCustomProperties()` method may be called during runtime to collect default values for settings.

For a description of the arguments and usage of these methods, refer to the HATS API References (Javadoc) for the HCustomProperty class. See “Using the API documentation (Javadoc)” on page 2.

Chapter 9. Using the HATS bidirectional API

Note: There is no bidirectional support for global variable overrides and Light pen (attention) and Light pen (selection) host components.

If you create HATS applications that use bidirectional (Arabic or Hebrew) code pages, and you add business logic or create your own custom components or widgets, you can use the bidirectional API to handle the recognition of host components and the presentation of widgets in the rich client transformation. This chapter describes this API. Before using the material in this chapter you should be familiar with the bidirectional concepts described in *HATS User's and Administrator's Guide*.

Note: You can find documentation on the VisualText component in the Rational SDP Help under the topic **SWT Bidi Extensions**.

Data Conversion APIs

Two APIs for handling text conversion from visual to logical and vice versa are included in the HostScreen class. You can use these APIs when creating custom widgets and components to handle the extraction of data.

ConvertVisualToLogical

```
public java.lang.String ConvertVisualToLogical(java.lang.String
inputBuffer, boolean isleft-to-rightVisual,
boolean isleft-to-rightImplicit)
```

Converts the given string from visual to implicit format and returns the implicit format of the string.

inputBuffer
The input string in visual format.

isleft-to-rightVisual
If true, inputBuffer is in visual left-to-right form.

isleft-to-rightImplicit
If true, the output buffer is in implicit left-to-right form.

ConvertLogicalToVisual

```
public java.lang.String ConvertLogicalToVisual(java.lang.String
inputBuffer, boolean isleft-to-rightImplicit,
boolean isleft-to-rightVisual)
```

Converts the given string from implicit to visual format and returns the visual format of the string.

inputBuffer
The input string in implicit format.

isleft-to-rightImplicit
If true, inputBuffer is in implicit left-to-right form.

isleft-to-rightVisual
If true, the output buffer is in visual left-to-right form.

Global Variable APIs

There are two getter methods that you can use to get the value of global variables. Using these methods you can get the global variable value either in implicit format or in visual format. These two methods are in class com.ibm.hats.common.BaseInfo.

getGlobalVariable

```
public IGlobalVariable getGlobalVariable(String name, boolean
createIfNotExist,boolean bidiImplicit)
```

Gets the named global variable, optionally creating it if it does not already exist.

createIfNotExist
Indicates whether or not to create a nonexistent global variable.

bidiImplicit
Indicates whether to get the global variable value in implicit format if `true`, or in visual format if `false`.

getSharedGlobalVariable

```
public IGlobalVariable getSharedGlobalVariable(String name, boolean
createIfNotExist,boolean bidiImplicit)
```

Gets the named shared global variable, optionally creating it if it does not already exist.

createIfNotExist
Indicates whether or not to create a nonexistent global variable

bidiImplicit
Indicates whether to get the global variable value in implicit format if `true`, or in visual format if `false`.

BIDI OrderBean

You can use the methods of the BIDI OrderBean for the correct display of bidirectional data. It contains the following parameters:

BidiString
String. Contains bidirectional text

FromTextVisual
Boolean. Indicates whether the source bidirectional text is visual. Default is `true`.

FromOriLTR
Boolean. Indicates whether the orientation of the source bidirectional text is LTR. Default is `true`.

ToTextVisual
Boolean. Indicates whether the target bidirectional text is visual. Default is `true`.

ToOriLTR
Boolean. Indicates whether the orientation of the target bidirectional text is LTR. Default is `true`.

NeedShape
Boolean. Indicates whether bidirectional text is Arabic text and whether it needs shaping. Default is `false`.

CharSet
String. Defines the character encoding for the JSP.

NumShape
String. Defines the numerals shaping method. Default is `Nominal`.

SymSwap
Boolean. Indicates whether symmetric swapping is on. Default is `false`.

BIDI OrderBean methods

setBidiString
```
public void setBidiString (String BdString)
```
Sets the bidirectional text to be reordered to the given string. The only parameter is **BdString**, which is the bidirectional string that needs reordering.

getBidiString
```
public String getBidiString ()
```
Gets the bidirectional text. Returns the bidirectional string that needs reordering.

setFromTextVisual
```
public void setFromTextVisual (boolean on)
```
Sets the source bidirectional text type as visual. The only parameter is **on**. If `true`, defines this source bidirectional text as visual. If `false`, defines this source bidirectional text as implicit.

setFromOriLTR
```
public void setfromOriLTR (boolean on)
```
Sets the source bidirectional text orientation as LTR. The only parameter is **on**. If `true`, defines this source bidirectional text as LTR. If `false`, defines this source bidirectional text as RTL.

setToTextVisual
```
public void setToTextVisual (boolean on)
```
Sets the target bidirectional text type as visual. The only parameter is **on**. If `true`, defines this target bidirectional text as visual. If `false`, defines this target bidirectional text as implicit.

setToOriLTR
```
public void setToOriLTR (boolean on)
```
Sets the target bidirectional text orientation as LTR. The only parameter is **on**. If `true`, defines this target bidirectional text as LTR. If `false`, defines this target bidirectional text as RTL.

setEncoding
```
public void setEncoding (String CharSet)
```
Sets the encoding character set. The only parameter is **CharSet**, which is a character-encoding name.

setNeedShape
```
public void setNeedShape (boolean on)
```

Sets the need to perform shaping. The only parameter is **on**. If `true`, indicates the need to perform shaping on the bidirectional text.

Order `public void Order ()`

Performs the ordering of the bidirectional text. There are no parameters.

CompressLamAlef

```
public String CompressLamAlef(String input,boolean direction)
```

Returns a string in which a Lam character followed by an Alef character is replaced by one LamAlef character. Parameters are:

- Direction. If `true`, indicates input text is in visual form. If `false`, input text is in implicit form.
- Input. An input string containing LamAlef characters to be compressed.

ExpandLamAlef

```
public String ExpandLamAlef(String input,boolean direction)
```

Returns a string in which a Lam Alef character is replaced by a Lam followed by one Alef character. Parameters are:

- Direction. If `true`, indicates input text is in visual form. If `false`, input text is in implicit form.
- Input. An input string containing LamAlef characters to be expanded.

setNumerals

```
public void setNumerals(String NumShape)
```

Sets the numerals shape of the output buffer. The only parameter is:

- NumShape. A string that takes one of three values:
  - NOMINAL. Numerals are in Latin format.
  - CONTEXTUAL. Numerals follow numbers.
  - NATIONAL. Numerals are in National format.
- Input. An input string containing LamAlef characters to be expanded.

setSymSwap

```
public void setSymSwap (boolean on)
```

Sets the Symmetric swapping option with Visual RTL orientation. The only parameter is **on**. If `true`, symmetric swapping is enabled for swapping characters in RTL screens. If `false` (the default), symmetric swapping is disabled for swapping characters in RTL screens.

ShapeArabicData

```
public String ShapeArabicData(String strInBuffer,boolean
isLTRVisual, boolean EnableNumSwap)
```

Returns a string in which Arabic data is shaped. Parameters are:

- strInBuffer. The bidirectional string that needs shaping.
- isLTRVisual. An input string containing LamAlef characters to be expanded. If `true`, bidirectional string is left to right visual. If `false`, bidirectional string is right to left visual.
- EnableNumSwap. If `true`, enable Numeric swapping. If `false`, disable numeric swapping.

DeshapeArabicData

```
public String DeshapeArabicData (String strInBuffer,boolean
isLTRVisual,boolean EnableNumSwap)
```

Returns a string in which Arabic data is deshaped. Parameters are:

- strInBuffer. The bidirectional string that needs deshaping.
- isLTRVisual. If `true`, bidirectional string is left to right visual. If `false`, bidirectional string is right to left visual.
- EnableNumSwap. If `true`, enable numeric swapping. If `false`, disable numeric swapping.

ConvertLogicalToVisual

```
 public java.lang.String ConvertLogicalToVisual(java.lang.String
inputBuffer, boolean isLTRimplicit,
boolean isLTRVisual)
```

Converts the given string from implicit to visual format and returns the visual format of the string. Parameters are:

- InputBuffer. The input string in implicit format.
- isLTRimplicit. If `true`, inputBuffer is in implicit left-to-right form.
- isLTRVisual. If `true`, the output buffer is in visual left-to-right form.

ConvertVisualToLogical

```
 public java.lang.String ConvertVisualToLogical(java.lang.String
inputBuffer, boolean isLTRVisual,
boolean isLTRimplicit)
```

Converts the given string from visual to implicit format and returns the implicit format of the string. Parameters are:

- InputBuffer. The input string in visual format.
- isLTRimplicit. If `true`, the output buffer is in implicit left-to-right form.
- isLTRVisual. If `true`, inputBuffer is in visual left-to-right form.

Appendix A. HATS Toolkit files

When you use HATS Toolkit to build your project, files for each component of the project are created. This appendix tells you where the file is located on your system, how to view and edit the source for the file, and describes the tags that make up each file.

Note: Use the HATS Toolkit editors if you edit these source files.

All of the files you create with HATS Toolkit are stored on your system in one or more workspaces managed by your Rational SDP program, such as Rational Application Developer. You can choose your workspace directory, and you can have more than one. Refer to the information provided with Rational SDP for information about choosing your workspace.

All of the file locations in this appendix refer to the relative path from the directory named for your project, which will be created within your workspace.

Application file (.hap)

The application file contains XML tags that define the settings you select when you create the project.

The application (.hap) file is stored in the *project_name*/profiles directory, where *project_name* is the name you gave the project when you created it. The application (.hap) file for a HATS EJB project is stored in the *project_name*/ejbModule directory. To view and edit the source of the application file for your HATS project, expand your project in the **HATS Projects** view and double-click **Project Settings** to open the project editor. You can view the source by clicking on the **Source** tab.

You can modify the application file using any of the tabs in the project editor. HATS Toolkit updates the affected information on other tabs when you make changes on any tab.

<application> tag

The <application> tag is the enclosing tag for the project.

The attributes of the <application> tag are:

active This attribute is not used by HATS. It is contained here for compatibility with HATS Limited Edition.

configured
This attribute is not used by HATS. It is contained here for compatibility with HATS Limited Edition.

description
Specifies the description you enter when you create a project.

template
Specifies the name of the template you selected for the project when you created the project. The default template is predefined.rcp.templates.Modern.

<connections> tag

The <connections> tag is a container for all the connection tags that define connections for this project.

The attributes of the <connections> tag are:

default
Specifies the name of the default connection. The default connection, which is created using the connection values that you specify in the New HATS Project wizard, defaults to the name of main.

<connection> tag

The <connection> tag identifies a connection defined for the project and points to the connection (.hco) file that defines the connection.

The attributes of the <connection> tag are:

name Specifies the name you entered when you created the connection.

<eventPriority> tag

The <eventPriority> tag is the enclosing tag for the screen events you defined for the project. The order of the event tags within the <eventPriority> tag is the order in which screen events are checked when a new host screen is encountered. This tag has no attributes.

<event> tag

The <event> tag specifies a screen event that you defined for the project.

The attributes of the <event> tag are:

enabled
Specifies whether the screen event's screen recognition criteria should be checked when a new host screen is encountered. Valid values are `true` and `false`. The default value is `true`.

name Specifies the name you gave the screen event when you defined it. If you store a screen event file under a folder (or group), the name of the folder is prepended to the name of the file.

type Specifies that this is a screen combination event. The available attribute is **screenCombination**.

<classSettings> tag

The <classSettings> tag is the enclosing tag for the Java classes you include in the project. This tag has no attributes.

<class> tag

The <class> tag specifies a class whose attributes are defined in the enclosed <setting> tags.

The attributes of the <class> tag are:

name Specifies one of the following Java classes:

- com.ibm.hats.common.AppletSettings
- com.ibm.hats.common.ApplicationKeypadTag

- com.ibm.hats.common.ClientLocale
- com.ibm.hats.common.DBCSSettings
- com.ibm.hats.common.DefaultConnectionOverrides
- com.ibm.hats.common.DefaultGVOverrides
- com.ibm.hats.common.HostKeypadTag
- com.ibm.hats.common.KeyboardSupport
- com.ibm.hats.common.OIA
- com.ibm.hats.common.RuntimeSettings
- com.ibm.hats.rcp.transform.widgets.*name*
- com.ibm.hats.rcp.ui.views.ToolBarSettings
- com.ibm.hats.transform
- com.ibm.hats.transform.components.*name*
- com.ibm.hats.transform.DefaultRendering

<setting> tag

The <setting> tag specifies the settings associated with the class in which the <setting> tag is enclosed. The <setting> tag contains **name** and **value** pairs for each of the classes. The following sections described the **name** and **value** pairs for each of the classes.

com.ibm.hats.common.AppletSettings

For the com.ibm.hats.common.AppletSettings class, name specifies a customizable setting for the rich client application asynchronous update function:

enable
: If `true`, enables the rich client application asynchronous update function. The default is **true**.

com.ibm.hats.common.ApplicationKeypadTag

For the com.ibm.hats.common.ApplicationKeypadTag class, name specifies a customizable setting:

show If `true`, shows a keypad in the application.

showDefault
: If `true`, shows a key in the application keypad to change the presentation to the default transformation.

showDisconnect
: If `true`, shows a key in the application keypad to disconnect from the host.

showKeyboardToggle
: If `true`, shows a key in the application keypad for toggling display of a host keyboard.

showRefresh
: If `true`, shows a key in the application keypad to refresh the browser window contents using the original transformation, and restore the input fields to their original value.

showReverse
: If `true`, shows a key in the application keypad for bidirectional support.

com.ibm.hats.common.ClientLocale

For the com.ibm.hats.common.ClientLocale class, name is always `locale`. The value for the locale setting specifies the language to be used to display button captions and messages. Value can be one of the following. The default is **accept-language**.

Characters that identify the country code of the locale

ar Arabic

cs Czech

de German

en English

es Spanish

fr French

hu Hungarian

it Italian

ja Japanese

ko Korean

pl Polish

pt_BR Brazilian Portuguese

ru Russian

tr Turkish

zh Simplified Chinese

zh_TW
Traditional Chinese

accept-language
The language is acquired from the `Accept-Language` HTTP header of the user's browser.

com.ibm.hats.common.DBCSSettings

For the com.ibm.hats.common.DBCSSettings class, there are three settings, autoConvertSBCStoDBCS, setATOKDefaultModetoRoman, and showUnprotectedSISOSpace.

- Valid values for the **autoConvertSDBCtoDBCS** attribute are:

  true Automatically convert single byte characters to double byte characters for 3270 and 3270E G-type or 5250 G-type and J-type fields.

  false Do not automatically convert single byte characters to double byte characters for 3270 and 3270E G-type or 5250 G-type and J-type fields.

  The default is **false**. For more information, see the section Project settings editor in the *HATS User's and Administrator's Guide*.
- Valid values for the **setATOKDefaultModetoRoman** attribute are:

  true Set the ATOK default input mode as Roman.

  false Set the ATOK default input mode as Hanji.

  The default is **false**.
- Valid values for the **showUnprotectedSISOSpace** attribute are:

true Show any unprotected Shift In or Shift Out characters as a space.

false Do not use a space to show unprotected Shift In or Shift Out characters.

The default is **true**. For more information, see the section Project settings editor in the *HATS User's and Administrator's Guide*.

com.ibm.hats.common.DefaultConnectionOverrides

For the com.ibm.hats.common.DefaultConnectionOverrides class, there is always at least one <setting> tag with a name attribute of `allowAll`. This <setting> tag indicates the chosen default security policy regarding the overriding of connection parameters. Any exceptions to the chosen security policy for connection overrides are recorded with additional <setting> tags, with the name attribute set to the name of the exceptional connection parameter.

Valid values for the name attributes are:

true The end user can override the named connection parameter. If the named connection parameter is `allowAll`, this means that all unnamed connection parameters may be overridden with clients requests.

false The end user can not override the named connection parameter. If the named connection parameter is `allowAll`, this means that no unnamed connection parameters may be overridden

The default for the `allowAll` setting is **false**.

com.ibm.hats.common.DefaultGVOverrides

For the com.ibm.hats.common.DefaultGVOverrides class, there is always at least one <setting> tag with a name attribute of `allowAll`. This <setting> tag indicates the chosen default security policy regarding the overriding of global variables. Any exceptions to the chosen security policy are recorded with additional <setting> tags, with the name attribute set to hatsgv_*variableName* for regular global variable exceptions, or hatssharedgv_*variableName* for shared global variable exceptions.

Valid values for the name attributes are:

true The end user can override the named connection parameter. If the named connection parameter is `allowAll`, this means that all unnamed connection parameters may be overridden with clients requests.

false The end user can not override the named connection parameter. If the named connection parameter is `allowAll`, this means that no unnamed connection parameters may be overridden

The default for the `allowAll` setting is **false**.

com.ibm.hats.common.HostKeypadTag

For the com.ibm.hats.common.HostKeypadTag class, name specifies a customizable setting:

show If `true`, shows a host keypad in the application.

showAltView
If `true`, shows an AltView key in the host keypad.

showAttention
If `true`, shows an ATTN key in the host keypad.

showClear
If `true`, shows a CLEAR key in the host keypad.

showEnter
If `true`, shows an Enter key in the host keypad.

showF1 – showF24
If `true`, shows a Function key with the corresponding number in the host keypad.

showFieldExit
If `true`, shows a Field Exit key in the host keypad.

showFieldMinus
If `true`, shows a Field Minus key in the host keypad.

showFieldPlus
If `true`, shows a Field Plus key in the host keypad.

showHelp
If `true`, shows a Help key in the host keypad.

showPA1
If `true`, shows a PA1 key in the host keypad.

showPA2
If `true`, shows a PA2 key in the host keypad.

showPA3
If `true`, shows a PA3 key in the host keypad.

showPageDown
If `true`, shows a Page Down key in the host keypad.

showPageUp
If `true`, shows a Page Up key in the host keypad.

showPrint
If `true`, shows a PRINT key in the host keypad for printing output.

showReset
If `true`, shows a RESET key in the host keypad.

showSystemRequest
If `true`, shows a SYSREQ key in the host keypad.

style Specifies how keys defined with value=`true` are displayed in the host keypad. Valid values are `buttons` or `links`. The default is **buttons**.

com.ibm.hats.common.KeyboardSupport

For the com.ibm.hats.common.KeyboardSupport class, name specifies a customizable setting:

enable
Specifies whether keyboard support is available in the project. When keyboard support is enabled, end users can use the physical keyboard keys to interact with the host. The end user can press certain physical keys that have been mapped to host aid keys, such as the F1, SYSREQ, RESET, or

ATTN keys. The end user can toggle keyboard support to be disabled if he wants to use a mapped physical keyboard key to interact with the browser.

Note: This must be set to `true` to turn on the wizard that allows the HATS theme to change from the default emulator style to a modern application style.

initialState
If `true`, the initial state of the host keyboard is on (the user can interact with the application using the physical keyboard).

supportAllKeys
If `true`, all mapped keys are supported, regardless of what buttons or links are displayed. If `false`:

- If there are no recognized host functions displayed in the current page as buttons or links, support all mapped host functions.
- If there are any recognized host function buttons or links, support only those host functions.

com.ibm.hats.common.OIA

For the com.ibm.hats.common.OIA class, name specifies a customizable setting:

active If `true`, an operator information area (OIA) is visible in the project. The default is **true**.

Note: This must be set to `true` to turn on the wizard that allows the HATS theme to change from the default emulator style or to a modern application style.

appletActive
If `true`, an indicator is displayed in the OIA if asynchronous update support is enabled. The default is **false**.

autoAdvanceIndicator
If `true`, displays in the OIA whether auto-advance is enabled, if it is supported by the browser. The default is **false**.

bidiControls
If `true`, displays in the OIA the current bidirectional controls to indicate editing status, if they are supported by the browser. The default is **true**.

cursorPosition
If `true`, displays in the OIA the absolute cursor position for the host, such as 1391. The default is **false**.

cursorRowColumn
If `true`, displays in the OIA the row and column of the host cursor, such as 18/031. The default is **true**.

fieldData
If `true`, displays in the OIA field extended data, such as numeric only or field exit required. The default is `false`.

inputInhibited
If `true`, displays in the OIA whether the keyboard is locked, preventing input from the keyboard. The default is **true**.

insertMode
If `true`, displays in the OIA whether overwrite mode is enabled, if it is supported by the browser. The default is **true**.

msgWaiting
If true, displays an indicator when the host system has one or more message for the session. The setting is applicable only for 5250 host systems.

sslCheck
If `true`, displays in the OIA whether the Host On-Demand connection is SSL secured. The default is **true**.

systemWait
If `true`, displays in the OIA whether the system is locked while waiting for data to be returned. The default is **true**.

typeAheadField
If `true`, displays the type-ahead field in the OIA. This field displays the type-ahead data as the user enters it, but the field cannot be directly edited. This setting is only effective when type-ahead support is enabled. See Enable type-ahead support in the *HATS User's and Administrator's Guide*. The default is **false**.

com.ibm.hats.common.RuntimeSettings

For the com.ibm.hats.common.RuntimeSettings class, name specifies a customizable setting:

autoEraseFields
Specifies whether modified input fields should have `[erasefld]` applied before modified data is entered into the field. The default value is **true**. If the value is set to `false`, space characters may be used to replace data already entered in the field by the host.

Notes:

1. Any host field that is rendered as multiple input fields will not be automatically cleared. For example, long host fields that wrap from one line to the next are rendered as multiple input fields and will not be automatically cleared before updating.
2. This setting can only be specified at the project level. It cannot be specified for a single transformation.

enableArrowKeyNavigation
When set to `true`, fields can be navigated with the keyboard arrow keys on the rendered host screen. The default is **false**.

enableAutoAdvance
Specifies whether the cursor moves to the next input field when located at the end of an input field; that is, when the input field is entirely filled in. When `true`, the cursor will move to the next input field when located at the end of an input field. When `false`, the cursor does not move to the next input field unless the user explicitly moves it. The default is **false**.

enableAutoTabOn
Specifies whether the tab key will move the cursor to the next input field when the cursor reaches the end of an input field; that is the input field is entirely filled in. When set to `true`, based on the order of presentation field in the browser, the tab key will move cursor in the current field to the next field when the position of the cursor is at the end of the current field. When set to `false`, the tab key does not move to the next input field unless the user explicitly moves it. The default is **false**.

enableOverwriteMode
If `true`, text entered into an input field overwrites text at the cursor

position one character at a time. If `false`, text entered into an input field is inserted at the cursor position pushing existing text ahead. The user can toggle from this initial setting using the Insert key. The default is **true**.

enableTypeAhead
When set to `true`, the user can begin typing data intended for input fields on the next screen (or screens) sent by the host, before they are received and processed by HATS. As the next screen (or screens) are received, HATS sends the previously typed data (type-ahead data) including any keys that submit the input to the host. See Enable type-ahead support in the *HATS User's and Administrator's Guide*. The default is **false**.

includeLabelsInTabOrder
If `true`, protected read-only labels are included in the default tab order. By default, the tabbing order only includes the input fields on the panel. This setting indicates that read-only labels should also be included.

selectAllOnFocus
If `true`, all text in a field is selected when the field receives focus, which is typical behavior for a Web application. If `false`, no text is selected when the field receives focus which is typical behavior for a terminal emulator.

Notes:

1. For Web applications:
   - The default is **true**.
   - This setting does not affect the **enableOverwriteMode** setting behavior.
   - This setting is only valid when Internet Explorer is used as the browser for the application.
2. For rich client applications:
   - The default is **false**.
   - When selected, this setting functions like the **enableOverwriteMode** setting in that characters are overwritten as a user types into the field.
   - Text is selected only when the keyboard is used to tab into the field. Text is not selected when clicking the mouse in the field.

suppressUnchangedData
If `true`, disables all fields whose contents are the same as when the form was rendered. If `false`, sends any field contents received from the browser to the host even if the current presentation space contents are identical for that field. The default is **false**.

com.ibm.hats.rcp.transform.widgets.*name*

Refer to the *HATS User's and Administrator's Guide* for descriptions of widget settings.

com.ibm.hats.rcp.ui.views.ToolBarSettings

For the com.ibm.hats.rcp.ui.views.ToolBarSettings class, name specifies a customizable setting:

displayAs
Specifies how to display items on the main Transformation view toolbar. Valid values are `TEXT`, `IMAGE`, and `BOTH`, the default is **TEXT**.

show Specifies whether to show the main Transformation view toolbar. Valid values are `true` and `false`. The default is **true**.

com.ibm.hats.transform

For the com.ibm.hats.transform class, name specifies a customizable setting:

alternate
The value `DEFAULT` if an alternateRenderingSet is specified. Otherwise, unspecified.

alternateRenderingSet
Specifies the name of the rendering set to use for default rendering if nothing is found to render during transformation of a HATS component tag.

com.ibm.hats.transform.components.*name*

Refer to the *HATS User's and Administrator's Guide* for descriptions of component settings.

com.ibm.hats.transform.DefaultRendering

For the com.ibm.hats.transform.DefaultRendering class, name is always `applicationDefaultRenderingSetName`. The value specifies the name of the rendering set defined as the default rendering set for the project. The rendering set name specified on the value setting must match the value of the default attribute specified for the <defaultRendering> tag.

<textReplacement> tag

The <textReplacement> tag is the enclosing tag for any text replacement values you define in the project. This tag has no attributes.

<replace> tag

The <replace> tag specifies the text replacement values in a project.

Note: If you are using a bidirectional code page, refer to *HATS User's and Administrator's Guide*.

The attributes of the <replace> tag are:

caseSensitive
Specifies whether the case of text replacement values must match before text replacement occurs. Valid values are `true` and `false`. The default is `false`.

from Specifies the text you want to replace. The text on the **from** attribute must be enclosed in quotes.

to When replacing text with text or HTML coding (Web only), specifies the replacement string you want to insert in place of the value specified on the **from** attribute. The replacement string on the **to** attribute must be enclosed in quotes. If you want to replace the text with a button or a link, the code for the button or link must be added inside the quotes.

regularExpression
Specifies whether Java regular expression support is used as part of the text replacement algorithm. A regular expression is a pattern of characters that describes a set of strings. You can use regular expressions to find and modify occurrences of a pattern. Valid values are `true` and `false`. The default is `false`.

toImage
When replacing text with an image, specifies the path and name of the

image you want to insert in place of the value specified on the **from** attribute. The path and name of the image on the **toImage** attribute must be enclosed in quotes.

matchLTR
When using a bidirectional code page, specifies whether the value specified on the **from** attribute is replaced on left-to-right screens. Valid values are `true` and `false`. The default is `true`.

matchRTL
When using a bidirectional code page, specifies whether the value specified on the **from** attribute is replaced on right-to-left screens. Valid values are `true` and `false`. The default is `false`.

matchReverse
When using a bidirectional code page, specifies whether the value specified on the **from** attribute is replaced when the section of the screen in which it appears has been reversed from the original direction of the page. Valid values are `true` and `false`. The default is `false`.

Note: Care should be taken when using text replacement. Text replacement with a disparate number of characters in the strings can cause changes in the representation of the screen. Depending on the widget used for presenting a region of a screen, text on a line of the screen can be contracted, expanded, or forced to a new line.

<defaultRendering> tag

The <defaultRendering> tag is the enclosing tag for all rendering sets you define in the project.

The attribute of the <defaultRendering> tag is:

default
Specifies the name of the rendering set to use for default rendering in the project. The rendering set name specified on the default attribute *must* match the value specified for the value attribute of the class setting named com.ibm.hats.transform.DefaultRendering.

<renderingSet> tag

The <renderingSet> tag is the enclosing tag for rendering items defined in the rendering set.

The attributes of the <renderingSet> tag are:

name The name specified for the rendering set when it was created.

description
The description specified for the rendering set when it was created.

layout Indicates whether to use compact rendering, which eliminates unnecessary blanks in fields and text on the transformed screen. This attribute should only be used if you want your default rendering to be compacted. The only valid value for layout is COMPACT. By default, a rendering set does not specify this attribute and does not use compact rendering.

separated
Indicates whether to render the output using inline span tags to differentiate between fields and reduce the amount of HTML and blank

space on the transformed screen. This is the default for Web applications optimized for mobile devices. By default, a rendering set does not specify this attribute.

table Indicates whether to render the output in a table and preserve the layout of the original host screen. This is the default for Web applications not optimized for mobile devices.

<renderingItem> tag

The <renderingItem> tag is the enclosing tag for a specific rendering item.

The attributes of the <renderingItem> tag are:

componentIdentifier
The name of the rendering item used to coordinate component information with the transformation. The default setting is the name of the screen combination event.

associatedScreen
The name of the captured screen used to create this rendering item.

description
The description entered when the rendering item was created.

enabled
Indicates whether this rendering item is enabled. Reflects the state of the check box on the Rendering page of Project Settings.

endCol
The last column of the host screen to which this rendering item should be applied. -1 means the rightmost column of the host screen.

endRow
The last row of the host screen to which this rendering item should be applied. -1 means the bottom row of the host screen.

startCol
The first column of the host screen to which this rendering item should be applied.

startRow
The first row of the host screen to which this rendering item should be applied.

type The host component whose contents will be transformed. The attribute value is the full class name of the host component. There is no default value for this required attribute.

widget
The widget into which the host component will be transformed.

The following tags are also included in each specific rendering item:

componentSettings
The <componentSettings> tag is the enclosing tag for any settings modified for the component for this rendering item. This tag has no attributes.

setting
The <setting> tag is the enclosing tag for any settings modified for the component for this rendering item.

The attributes of the <setting> tag are:

name Specifies the name of a customizable setting for the component. The available settings depend on the component.

Refer to *HATS User's and Administrator's Guide* for descriptions of component settings.

value Specifies the value of a customizable setting for the component. The default values vary depending on the setting.

widgetSettings

The <widgetSettings> tag is the enclosing tag for any settings modified for the widget for this rendering item. This tag has no attributes.

setting

The <setting> tag is the enclosing tag for any settings modified for the widget for this rendering item.

The attributes of the <setting> tag are:

name Specifies the name of a customizable setting for the widget. The available settings depend on the widget.

Refer to *HATS User's and Administrator's Guide* for descriptions of widget settings.

value Specifies the value of a customizable setting for the widget. The default values vary depending on the setting.

textReplacements

The <textReplacements> tag is the enclosing tag for any text replacement specified for this rendering item. This tag has no attributes.

replace

The <replace> tag specifies the text replacement values for this rendering item.

The attributes of the <replace> tag are:

caseSensitive
Specifies whether the case of text replacement values must match before text replacement occurs. Valid values are `true` and `false`. The default is `false`.

from Specifies the text you want to replace. The text on the **from** attribute must be enclosed in quotes.

to Specifies the replacement string you want to insert in place of the value specified on the **from** attribute. The replacement string on the **to** attribute must be enclosed in quotes.

regularExpression
Specifies whether Java regular expression support is used as part of the text replacement algorithm. A regular expression is a pattern of characters that describes a set of strings. You can use regular expressions to find and modify occurrences of a pattern. Valid values are `true` and `false`. The default is `false`.

toImage
When replacing text with an image, specifies the path and name of the image you want to insert in place of the value specified on the **from** attribute. The path and name of the image on the **toImage** attribute must be enclosed in quotes.

matchLTR
When using a bidirectional code page, specifies whether the value specified on the **from** attribute is replaced on left-to-right screens. Valid values are `true` and `false`. The default is `true`.

matchRTL
When using a bidirectional code page, specifies whether the value specified on the **from** attribute is replaced on right-to-left screens. Valid values are `true` and `false`. The default is `false`.

matchReverse
When using a bidirectional code page, specifies whether the value specified on the **from** attribute is replaced when the section of the screen in which it appears has been reversed from the original direction of the page. Valid values are `true` and `false`. The default is `false`.

Note: Care should be taken when using text replacement. Text replacement with a disparate number of characters in the strings can cause changes in the HTML representation of the screen. Depending on the widget used for presenting a region of a screen, text on a line of the screen can be contracted, expanded, or forced to a new line.

<globalRules> tag

The <globalRules> tag is the enclosing tag for any global rules you define in the project. It has no attributes.

<rule> tag

The <rule> tag defines a global rule.

The attributes of the <rule> tag for project-level rules are the same as for screen customization-level global rules. However, when you create a project-level and a screen customization-level global rule using the same input field, the screen customization-level rule will have a higher priority. The <rule> tag attributes are:

associatedScreen
The name of a screen capture in the project, from which the global rule is defined.

description
The description entered when the global rule was created.

enabled
Indicates whether this global rule is enabled. Reflects the state of the check box on the Rendering page of Project Settings.

endCol
The last column of the host screen to which this global rule should be applied. -1 means the rightmost column of the host screen.

endRow
The last row of the host screen to which this global rule should be applied. -1 means the bottom row of the host screen.

name The name that will be shown in the list of global rules on the Rendering page of Project Settings.

startCol
The first column of the host screen to which this global rule should be applied.

startRow
The first row of the host screen to which this global rule should be applied.

transformationFragment
The name of the transformation fragment file associated with this global rule. This file contains the information specifying how to transform the host component. It will be included in a transformation if the appropriate input fields are present in the host screen.

type The pattern type component for this global rule, taken from the first page of the Create Global Rule wizard. The type can be one of the following:

com.ibm.hats.transform.components. InputFieldByTextPatternComponent
This pattern component recognizes input fields on the host screen based on text near the fields.

com.ibm.hats.transform.components. AllInputFieldsPatternComponent
This pattern component recognizes all input fields on the host screen.

com.ibm.hats.transform.components. InputFieldBySizePatternComponent
This pattern component recognizes input fields on the host screen based on the size of the input fields.

com.ibm.hats.transform.components. InputFieldByPositionPatternComponent
This pattern component recognizes input fields on the host screen by the field's position on the host screen.

The following tags are also included in each specific global rule:

componentSettings
The <componentSettings> tag is the enclosing tag for any settings defined for the pattern type component specified on the type attribute of the <rule> tag. This tag has no attributes.

setting
The <setting> tag is the enclosing tag for any settings defined for the pattern type component specified on the type attribute of the <rule> tag.

The attributes of the <setting> tag are:

name Specifies the name of a customizable setting for the pattern type component. The available settings depend on the component.

- For the com.ibm.hats.transform.components.InputFieldByTextPattern Component, the settings for the name attribute are:

caseSensitive
Specifies whether the case of the text on the text setting must match before the pattern is recognized. Valid values are `true` and `false`. The default is `true`.

immediatelyNextTo
Specifies which input fields you want to transform.Valid values are:

true Specifies that only the nearest input field should be transformed.

false Specifies that all input fields should be transformed.

The default is `false`.

location
Specifies where text in a protected field, as specified on the text setting, must be in relation to input fields for this global rule to be applied. Valid values are:

ABOVE
Specifies that the text must be above the input field.

BELOW
Specifies that the text must be below the input field.

LEFT Specifies that the text must be to the left of the input field.

RIGHT
Specifies that the text must be to the right of the input field.

The default is RIGHT.

text Specifies some text in a protected field of the host screen. Valid values are any text in a protected field on the host screen.

- For the com.ibm.hats.transform.components.AllInputFieldsPattern Component, there are no component settings.
- For the com.ibm.hats.transform.components.InputFieldBySizePattern Component, the setting for the **name** attribute is **fieldSize**.Valid values are the sizes of any input fields on the host screen.
- For the com.ibm.hats.transform.components. InputFieldByPositionPatternComponent, the setting for the **name** attribute is **enableFieldLength**. Valid values are `true` and `false`.

  Note: When **enableFieldLength** is specified, the entire field (as specified by the **fieldSize** attribute) must be within the defined region boundary in order for the field to be recognized. The region boundary is defined by the values for the **startRow**, **endRow**, **startCol** and **endCol** attributes.

Connection files (.hco)

Each connection that you define in a HATS project is represented by a connection file. The connection (.hco) files are stored in the *project_name*`/Connections` folder, where *project_name* is the name you gave the project when you created it. The default connection, which is created using the connection values that you specify in the New HATS Project wizard, is stored in main.hco.

<hodconnection> tag

The <hodconnection> tag begins the connection definition and specifies several characteristics for the connection.

The attributes of the <hodconnection> tag are:

certificateFile
Specifies the name of the file from which the project's SSL certificate was imported, if any.

codePage
Specifies the numeric value for the code page used on this connection. The default value is the value you selected when you created the project. Each connection can use a different code page. See the description of the codePageKey attribute for the code page numbers.

codePageKey
Specifies the usage key that corresponds to the numeric code page. The default value is KEY_US. Valid values for codePage and the location or usage key are:

Table 18. Code pages and usage keys

Code page	Usage key
037	KEY_BELGIUM KEY_BRAZIL KEY_CANADA KEY_NETHERLANDS KEY_PORTUGAL KEY_US
273	KEY_AUSTRIA KEY_GERMANY
274	KEY_BELGIUM_OLD
275	KEY_BRAZIL_OLD
277	KEY_DENMARK KEY_NORWAY
278	KEY_FINLAND KEY_SWEDEN
280	KEY_ITALY
284	KEY_SPAIN KEY_LATIN_AMERICA
285	KEY_UNITED_KINGDOM
297	KEY_FRANCE
420	KEY_ARABIC
424	KEY_HEBREW
500	KEY_MULTILINGUAL
803	KEY_HEBREW_OLD
838	KEY_THAI

Table 18. Code pages and usage keys (continued)

Code page	Usage key
870	KEY_BOSNIA_HERZEGOVINA KEY_CROATIA KEY_CZECH KEY_HUNGARY KEY_POLAND KEY_ROMANIA KEY_SLOVAKIA KEY_SLOVENIA
871	KEY_ICELAND
875	KEY_GREECE
924	KEY_MULTILINGUAL_ISO_EURO
930	KEY_JAPAN_KATAKANA
933	KEY_KOREA_EX
937	KEY_ROC_EX
939	KEY_JAPAN_ENGLISH_EX
1025	KEY_BELARUS KEY_BULGARIA KEY_MACEDONIA KEY_RUSSIA KEY_SERBIA_MONTEGRO
1026	KEY_TURKEY
1047	KEY_OPEN_EDITION
1112	KEY_LATVIA KEY_LITHUANIA
1122	KEY_ESTONIA
1123	KEY_UKRAINE
1137	KEY_HINDI
1140	KEY_BELGIUM_EURO KEY_BRAZIL_EURO KEY_CANADA_EURO KEY_NETHERLANDS_EURO KEY_PORTUGAL_EURO KEY_US_EURO
1141	KEY_AUSTRIA_EURO KEY_GERMANY_EURO
1142	KEY_DENMARK_EURO KEY_NORWAY_EURO
1143	KEY_FINLAND_EURO KEY_SWEDEN_EURO
1144	KEY_ITALY_EURO
1145	KEY_LATIN_AMERICA_EURO KEY_SPAIN_EURO
1146	KEY_UNITED_KINGDOM_EURO
1147	KEY_FRANCE_EURO
1148	KEY_MULTILINGUAL_EURO
1149	KEY_ICELAND_EURO

Table 18. Code pages and usage keys (continued)

Code page	Usage key
1153	KEY_BOSNIA_HERZEGOVINA_EURO KEY_CROATIA_EURO KEY_CZECH_EURO KEY_HUNGARY_EURO KEY_POLAND_EURO KEY_ROMANIA_EURO KEY_SLOVAKIA_EURO KEY_SLOVENIA_EURO
1154	KEY_BELARUS_EURO KEY_BULGARIA_EURO KEY_MACEDONIA_EURO KEY_RUSSIA_EURO KEY_SERBIA_MONTEGRO_EURO
1155	KEY_TURKEY_EURO
1156	KEY_LATVIA_EURO KEY_LITHUANIA_EURO
1157	KEY_ESTONIA_EURO
1158	KEY_UKRAINE_EURO
1160	KEY_THAI_EURO
1166	KEY_KAZAKHSTAN_EURO
1364	KEY_KOREA_EURO
1371	KEY_ROC_EURO
1388	KEY_PRC_EX_GBK
1390	KEY_JAPAN_KATAKANA_EX_EURO
1399	KEY_JAPAN_ENGLISH_EX_EURO

connecttimeout
Specifies the time that HATS attempts to connect to a host. Specify a number of seconds between 1 and 2147483647. The initial default is 120 seconds.

description
Specifies the description for the connection when it was created.

disableFldShp
When using a bidirectional code page, specifies whether you want Arabic data in password fields submitted to the host in isolated form or in shaped form. Valid values are `true` and `false`. There is no initial default.

disableNumSwapSubmit
When using a bidirectional code page, specifies whether you want to disable entry of Arabic-Western numbers, that is, allow entry of only Arabic-Indic numbers in RTL screens. Do this so that, when submitted, all numbers are submitted as Arabic-Western numbers. Valid values are `true` and `false`. There is no initial default.

disconnecttimeout
Specifies the time that HATS attempts to disconnect from a host. Specify a number of seconds in the range of 1-2147483647. The initial default is 120 seconds.

enableScrRev
When using a bidirectional code page, specifies which pages of an application should display a **Screen Reverse** button to enable users to reverse the direction of displayed text and input fields. Valid values are:

(blank)
The **Screen Reverse** button is not placed on any screens.

Customized
The **Screen Reverse** button is placed on screens that match a screen customization and on screens that do not match a screen customization, in other words, on all screens. There is no option to place the **Screen Reverse** button only on screens that match a screen customization.

Non-customized
The **Screen Reverse** button is placed only on screens that do not match a screen customization.

There is no initial default.

host Specifies the name of the host to which the connection is made.

hostSimulationName
Specifies the name of the host simulation trace file to use instead of a live connection.

LUName
Valid only on enhanced 3270 sessions (TNEnhanced="true"). Sets the LUName property, which is the LU name used during enhanced negotiation. Values are in string format. Maximum length of LUName is 17 characters. There is no default. To configure print support for your 3270 HATS project, you must specify that the host type is 3270E. When you add the LUName parameter to the list of connection settings, do not use the printer LU name; use the name of your display LU or a pool of display LUs.

LUNameSource
Valid only on enhanced 3270 sessions (TNEnhanced="true"). Specifies the source of the LU name for the connection. Valid values are:

automatic
The LU name is automatically assigned when the connection is established.

prompt
Prompt the end user for the LU name. If pooling is enabled, prompt should not be used.

session
The LU name is defined using an HTTP session variable. The LUName attribute names the session variable. If pooling is enabled, session should not be used.

value The LU name is defined on the LUName attribute.

There is no initial default.

port Specifies the number of the port through which the connection to the host is made. The valid range for ports is 0–65535. The initial default is 23.

screenSize
Specifies the number of rows and columns that the host terminal displays. Valid values for screenSize are:

- 2=24x80
- 3=32x80
- 4=43x80
- 5=27x132
- 6=24x132 (VT only)

The initial default screen size is 24 x 80.

sessionType
Specifies the type of terminal the host terminal displays. Valid values for type are:

- 1=3270
- 2=5250
- 3=VT

The initial default is 3270.

singlelogon
When user lists are defined in the project, specifies whether a user ID can be used more than once at a time. Valid values are:

true The user ID can be used only once at a time.

false The user ID can connect multiple times simultaneously.

The initial default is `false`.

SSL Specifies whether SSL is enabled. Valid values are:

true SSL is enabled for the project.

false SSL is not enabled for the project.

TNEnhanced
Valid only on 3270 connections. Specifies whether the connection is a TN3270E connection. Valid values are `true` and `false`. The initial default is `true`.

VTTerminalType
Valid only on VT connections. Indicates the type of VT terminal. Valid values are:

- 1=VT420_7
- 2=VT420_8
- 3=VT100
- 4=VT52

WFEnabled
Valid only on 5250 connections. Specifies whether the connection is a 5250W connection. Valid values are `true` and `false`. The initial default is `false`.

workstationID
Valid only on 5250 and 5250W connections. When the workstationIDSource attribute is set to either session or value, specifies the HTTP session variable or the workstation ID for the connection. There is no initial default.

workstationIDSource
Valid only on 5250 and 5250W connections. Specifies the source of the workstation ID for the connection. Valid values are:

automatic
The workstation ID is automatically assigned when the connection is established.

prompt
Prompt the end user for the workstation ID. If pooling is enabled, prompt should not be used.

session
The workstation ID is defined using an HTTP session variable. The workstationID attribute names the session variable. If pooling is enabled, session should not be used.

value The workstation ID is defined on the workstationID attribute.

There is no initial default.

<otherParameters> tag

The <otherParameters> tag specifies additional Host On-Demand session parameters.

Host On-Demand session parameters supported by HATS include:

ENPTUI
Determines whether a project with a connection to a 5250 host can use display data stream (DDS) keywords for the Enhanced Non-Programmable Terminal User Interface (ENPTUI). Valid values are `true` and `false`. The default value is `false`.

Lamalef
Sets the LamAlef property, which determines whether LamAlef should be expanded or compressed. This property applies to Arabic sessions only. Values are in string format. Valid values are:

- LAMALEF_ON
- LAMALEF_OFF

The default value is LAMALEF_OFF.

numeralShape
Sets the numeralShape property. This property applies to bidirectional sessions only. Values are in string format. The default value is NOMINAL.

numericSwapEnabled
Sets the Numeric swapping property. This property applies to Arabic 3270 sessions only. Valid values are `true` and `false`. The default value is `true`.

roundTrip
Sets the roundTrip property. This property applies to bidirectional sessions only. Values are in string format. Valid values are:

- ROUNDTRIP_ON
- ROUNDTRIP_OFF

The default value is ROUNDTRIP_ON.

SecurityProtocol
Sets the SecurityProtocol property, which indicates whether to use the TLS v1.0 protocol or the SSL protocol for providing security. Values are in string format. The default value is TLS.

SSLServerAuthentication
Sets the SSLServerAuthentication property, which indicates whether SSL server authentication is enabled. Valid values are `true` and `false`. The default value is `false`.

symmetricSwapEnabled
Sets the symmetric swapping property. This property applies to Arabic 3270 sessions only. Valid values are `true` and `false`. The default value is `true`.

textOrientation
Sets the textOrientation property. This property applies to bidirectional sessions only. Values are in string format. Valid values are:

- LEFT_TO_RIGHT
- RIGHT_TO_LEFT

The default value is LEFT_TO_RIGHT.

ThaiDisplayMode
Sets the Thai display mode property. This property applies to Thai sessions only. Values are in string format. The default value is THAI_MODE_5.

workstationID
Sets the workstationID property, which is used during enhanced negotiation for 5250. Values are in string format. All lowercase characters are converted to uppercase. There is no default value.

Kerberos ticket support

To enable support for Kerberos tickets, for rich client projects using 5250 only, add the following lines inside the `<otherParameters>` tag in the application.hap file:

```
<parameter name="ssoEnabled" value="true"/>

<parameter name="ssoType" value="ssoAcquireKerberosTicket"/>
```

<classSettings> tag

The <classSettings> tag is the enclosing tag for the Java classes you include in the connection definition.

<class> tag

The <class> tag specifies a class whose attributes are defined in the enclosed <settings> tags.

The attributes of the **<class>** tag are:

name Specifies one of the following Java classes:

- com.ibm.hats.common.HATSPrintSettings
- com.ibm.hats.common.NextScreenSettings

The class names on the name attribute must be enclosed in quotes.

<setting> tag

The <setting> tag specifies the settings associated with the class in which the <setting> tag is enclosed.

The attributes of the <setting> tag are:

name Specifies the name of a customizable setting for the class defined by the name attribute of the <class> tag. The available settings depend on the class.

For the com.ibm.hats.common.HATSPrintSettings class, the customizable settings are:

printFontName
Specifies the font in which you want your output printed. Valid values depend on the value of the codePage attribute.

printNumSwapSupport
Specifies whether numeric swapping is enabled. This property applies to Arabic 3270 sessions only, when printRTLSupport is enabled. English numerals are replaced by Arabic numerals in right-to-left screens and Arabic numerals are replaced by English numerals in right-to-left Screens. Valid values are `true` and `false`. The default value is `true`.

printOrientation
Specifies how your printed output is positioned on the page. Valid values for printOrientation are:

PDF_ORIENTATION_PORTRAIT
Orients the paper vertically.

PDF_ORIENTATION_LANDSCAPE
Rotates the paper 90 degrees clockwise.

printPaperSize
Specifies the size of the paper on which to print your output. Valid values for printPaperSize are:

ISO_A3
ISO/DN & JIS A4, 297 x 420 mm

ISO_A4
ISO/DN & JIS A4, 210 x 297 mm

ISO_A5
ISO/DN & JIS A4, 148 x 210 mm

ISO_B4
ISO/DN B4, 250 x 353 mm

ISO_B5
ISO/DN B5, 176 x 250 mm

JIS_B4
JIS B4, 257 x 364 mm

JIS_B5
JIS B5, 182 x 257 mm

ISO_C5
ISO/DN C5, 162 x 229 mm

ISO_DESIGNATED_LONG
ISO/DN Designated Long, 110 x 220 mm

EXECUTIVE
Executive, 7 1/4 x 10 1/2 in

LEDGER
Ledger, 11 x 17 in

NA_LETTER
North American Letter, 8 1/2 x 11 in

NA_LEGAL
North American Legal, 8 1/2 x 14 in

NA_NUMBER_9_ENVELOPE
North American #9 Business Envelope, 3 7/8 x 8 7/8 in

NA_NUMBER_10_ENVELOPE
North American #10 Business Envelope, 4 1/8 x 9 1/2 in

MONARCH_ENVELOPE
Monarch Envelope, 3 7/8 x 7 1/2 in

CONTINUOUS_80_COLUMNS
Data Processing 80 Columns Continuous Sheet, 8 x 11 in

CONTINUOUS_132_COLUMNS
Data Processing 132 Columns Continuous Sheet, 13 1/5 x 11 in

printRTLSupport
Specifies whether right-to-left print support is enabled. This property applies to Arabic 3270 sessions only. Bidirectional files can be either RTL or LTR files. Valid values are `true` and `false`. The default value is `true`.

printSupport
Specifies whether your project includes print capability. Valid values for printSupport are `true` and `false`. The initial default is `false`.

printSymSwapSupport
Specifies whether symmetric swapping is enabled; swapping characters are swapped in right-to-left screens. This property applies to Arabic 3270 sessions only, when printRTLSupport is enabled. Valid values are `true` and `false`. The default value is `true`.

printURL
Specifies the URL for a System i® Access for Web Printer Output window on a 5250 server. The default URL is http://*hostname*/webaccess/iWASpool, where *hostname* is the name of the 5250 server.

The customizable settings for the com.ibm.hats.common.NextScreenSettings class are:

default.appletDelayInterval
Specifies the maximum time (in milliseconds) that the server waits until a full host screen has arrived for a session running in asynchronous update mode. The initial default value is 400 milliseconds.

default.blankScreen
Specifies how to handle a blank screen received at connection startup. Valid values are:

normal
Display the blank screen.

sendkeys
Send the host key defined on the default.blankScreen.keys setting.

timeout
Wait for the connection to time out before issuing an error message.

The default is `normal`.

default.blankScreen.keys
Specifies the host key to send when default.blankScreen is set to sendkeys.

default.delayInterval
Specifies the maximum time, in milliseconds, that the server waits for the arrival of screen updates after the initial screen update. The initial default value is 1200 milliseconds.

default.delayStart
Specifies the minimum time (in milliseconds) that the server waits until the first full host screen has arrived after the host connection becomes ready. The initial default value is 2000 milliseconds.

nextScreenClass
Specifies a class that turns off the default, speed-optimized, algorithm in favor of accuracy. The class for the value attribute is com.ibm.hats.runtime.TimingNextScreenBean. As a result, screen transitions might be slower. The setting `default.delayInterval` is now the minimum amount of time (in milliseconds) per screen transition. The `default.delayInterval` value has a default of 1200 milliseconds, but you can customize it for your network and your host application. If you raise this value, remember that HATS waits at least this long for the host screen to settle.

oiaLockMaxWait
Specifies the maximum time (in milliseconds) that HATS should wait after the host screen has settled to ensure that the OIA system lock status has been released. The value can be in the range of 0–600000 milliseconds. The initial default value is 300000 milliseconds.

value The values for the settings are included in the description of the individual settings.

<poolsettings> tag

The <poolsettings> tag defines pooling parameters for the connection.

The attributes of the <poolsettings> tag are:

enabled
Specifies whether pooling is enabled for this connection. Valid values for enabled are `true` and `false`. The initial default is `false`.

maxbusytime
The number of seconds before a connection that is in use by an end user will be terminated. If you do not want active connections to end, set this field to -1. This setting is available for connections that have pooling enabled as well as for connections that have pooling disabled. For a connection with pooling enabled, the connection returns to the pool if the number of available connections in the pool is less than the minimum number of connections you specified to remain connected. Otherwise, this connection is discarded. For a connection with pooling disabled, the connection is discarded. Valid number of seconds is -1 or in the range of 60-2147483647. The default is -1 (no maximum busy time).

maxconnections
The maximum number of connections in the pool that can be active. This setting is available only for connections that have pooling enabled. Valid number of connections is in the range of 1-2147483647. The default is 1. When you reach the maximum specified and an additional request for a connection is received, HATS can either wait for the next available connection or create a new connection.

maxidletime
The number of seconds before a connection that is not in use by an end user will be terminated and removed from the pool. If you do not want inactive connections to end, set this field to -1. This setting is available only for connections with connection pooling enabled. The minimum number of connections you specify remain connected, whether or not they are used. Valid number of seconds is -1 or in the range of 60-2147483647. The default is -1 (no maximum idle time).

minconnections
The number of idle connections in the pool that remain connected. This setting is available only for connections that have pooling enabled and have the maxidletime before disconnection set to some value other than -1. Valid number of connections is between 0 and 2147483647. The default is 0 (do not keep connections connected).

overflowallowed
Whether a new, non-pooled connection should be created if the maximum limit of connections has been reached. If this value is false, you must specify the number of seconds to wait for a pooled connection to become available. If the time to wait elapses and a connection does not become available, HATS returns an error. If this value is true, a new, non-pooled connection will be created. When the end user finishes with this type of connection, it is not put back in the pool, but discarded.

waittimeout
The number of seconds to wait for a pooled connection to become available once the maximum limit of connections has been reached, and another request comes in. Valid number of seconds is between 0 and 2147483647, or -1 if you want to wait forever. The default is 120.

<userconfig> tag

The <userconfig> tag defines a user list for the connection. The tags and data within the <userconfig> tag are complex and can be corrupted by manual editing. To protect the integrity of your user list, do not manually edit the <userconfig> data. Instead, use the **User List** tab on the Connection editor to create or modify a user list. By default, a host connection does not specify this tag and does not have a user list.

Screen combination files (.evnt)

The screen combination files defines how a host screen is recognized, the actions HATS performs when a screen is recognized, how to define the end of the screen combination, and how to navigate between screens. The screen combination (.evnt) files are stored in the *project_name*/**profiles/events/screencombinations** directory. You can view and edit the source of the screen combination files by double-clicking on the name of the screen combination in the HATS Projects view to open the screen combination editor. The source for the file can be viewed by clicking on the **Source** tab. You can modify screen combination files using the **Begin Screen**, **Render**, **Navigation**, **End Screen**, **Actions**, **Text replacement**, or **Source** tabs in the editor. HATS Toolkit updates the affected information on other tabs when you make changes on any tab. The screen combination event files contain tags to define how a host screen is recognized and the actions and navigations that will occur when the host screen is recognized.

Screen combination adds several tags to those found in screen customization (.evnt) files.

<combinations> tag

This is the container for the combination information. It consists of a type attribute and a rendering item detailing the screen combination component.

type The type parameter determines how the screen transformation will be aggregated.

If the string value is set to *dynamic*, the screen transformation can add screens to the combined area while the user is using the screen transformation.

If the string value is set to *normal* or is missing, the individual screens compound prior to allowing the user to interact with the screen transformation. Rich Client screen combinations are limited to normal processing.

<enddescription> tag

This is the description for the screen criteria used to determine if the screen combination end screen has been reached. The tags and details match the description tag. It has an attribute associatedScreen for the screen associated with the end screen.

associatedScreen
This is the screen capture associated with the end screen.

<navigation> tag

The navigation contains the commands needed to move between screens to gather and place data. It consists of a screenUp and screenDown tag.

<screenUp> tag

The commands necessary to traverse to a screen backward in the combination. This is used to return data to the correct place in a screen combination. It can consist of keyPress, setCursor, and sendText tags.

<screenDown> tag

The commands necessary to traverse to a screen forward in the combination. This is used to create the screen combination view as well as return data to the correct place in a screen combination. It can consist of keyPress, setCursor, and sendText tags.

<keyPress> tag

This navigation command is the equivalent of a sendKey. It has a value attribute, which must be a valid HOD key command, for the HOD command to send.

value The value attribute for the keyPress tag should be a valid HOD key command.

<setCursor> tag

This navigation command allows cursor positioning on the screen. It has a row and a column attribute for the cursor positioning.

row The row attribute should be a 1-based integer that equates to a position on the screen. This positions the vertical component of the cursor position.

column
The column attribute should be a 1-based integer that equates to a position on the screen. This positions the horizontal component of the cursor position.

<sendText>

This navigation command is the equivalent of a sendKey. It has a value attribute, which must be valid text for the host field, for the text to send.

value The value attribute for the sendText tag should be valid text for the host field.

Screen customization files (.evnt)

The screen customization files define how a host screen is recognized, and also defines the actions HATS performs when a screen is recognized.

The screen customization (.evnt) files are stored in the *project_name*/**profiles/events/screencustomizations** directory. You can view and edit the source of the screen customization files by double-clicking on the name of the screen customization in the **HATS Project View** to open the screen customization editor. The source for the file can be viewed by clicking on the **Source** tab.

You can modify screen customization files using the **Screen Recognition Criteria**, **Actions**, **Text replacement**, or **Source** tabs in the editor. HATS Toolkit updates the affected information on other tabs when you make changes on any tab.

The screen customization event files contain tags to define how a host screen is recognized and the actions that will occur when the host screen is recognized.

<event> tag

Begins the definition of the screen customization. The event tag has the following attributes:

description
If you supplied a description of the screen customization when you created it, that description is defined in this attribute.

type For a screen customization, type is always screenRecognize. For combined screens, type is screenCombination.

<actions> tag

This is the enclosing tag for all of the actions defined for a screen customization. It has no attributes.

<apply> tag

Defines the action for applying a transformation. The attributes of the <apply> tag are:

applyGlobalRules
Specifies whether HATS should look for global rules on this host screen. Default is `true`.

applyTextReplacement
Specifies whether HATS should look for text replacements on this host screen. Default is `true`. See the <replace> tag for further information on how to use text replacement.

enabled
Indicates whether this action is enabled for use. The default is `true`.

immediateKeyset
Defines the host keys sent to the host immediately when pressed by the end user of your project. If you did not define any host keys to be sent to the host immediately, this attribute has an empty value.

template
Names the template file that surrounds the transformation being applied. If the default template is being used to surround the transformation, this attribute has an empty value.

transformation
Names the transformation file that is to be applied for this action.

<insert> tag

Defines the action for inserting a global variable or a string. The attributes of the <insert> tag are:

enabled
Indicates whether this action is enabled for use. The default is `true`.

row Defines the starting row on the host screen where the value is to be inserted.

col Defines the starting column on the host screen where the value is to be inserted.

source Specifies whether the value to be inserted is a string or the value of a global variable. Valid values are string and variable.

value Specifies either the string to be inserted onto the host screen or the name of a global variable from which the value is taken.

fill If the source of the value to be inserted is an indexed global variable, fill specifies whether the indices of the global variable are to be concatenated

and inserted at the specified position, or inserted into a rectangular region of the host screen. Valid values are concatenate and rectangular.

index If the source of the value to be inserted is an indexed global variable, index specifies the number of the index that is to be used as the value to be inserted onto the host screen.

shared
If the source of the value to be inserted is a global variable, shared specifies whether this global variable is shared between applications running in the same rich client environment.

<extract> tag

Defines the action for extracting a global variable. The attributes of the <extract> tag are:

enabled
Indicates whether this action is enabled for use. The default is `true`.

srow Defines the starting row on the host screen of the text being extracted.

erow Defines the ending row on the host screen of the text being extracted.

scol Defines the starting column on the host screen of the text being extracted.

ecol Defines the ending column on the host screen of the text being extracted.

name Specifies the name of the global variable to which the text is extracted. This can be an existing global variable or a new global variable.

overwrite
Specifies whether the text extracted is to overwrite the value of an existing global variable. Valid values are `true` and `false`.

indexed
Specifies whether the text extracted is a single string or a list of strings, where each string in the list corresponds to a single row of text from the extracted region. Valid values are true and false.

index If an existing global variable is indexed, this attribute specifies the index number to which the extracted value is to be written. The effect of this attribute is dependent on the value of the **overwrite** attribute. If `overwrite=true`, the extracted value overwrites the existing variable, starting at the specified index. If `overwrite=false`, the extracted value is inserted into the existing variable, beginning at the specified index.

shared
Shared specifies whether the global variable is shared between applications running in the same rich client environment.

<set> tag

Defines the action for setting a global variable. The attributes of the <set> tag are:

enabled
Indicates whether this action is enabled for use. The default is true.

name Specifies the name of the global variable being set. This can be an existing global variable or a new global variable.

shared
Specifies whether the global variable being set is shared between applications running in the same rich client environment.

type Specifies whether the value of the global variable being set comes from a fixed constant or a calculated value. Valid values are `string` and `calculate`.

value Specifies the value being assigned to the global variable.

overwrite
Specifies whether the value being set is to overwrite the value of an existing global variable. Valid values are `true` and `false`.

index If the value being set is being written to an existing indexed global variable, this attribute specifies the index number to which the value being set is written. The effect of this attribute is dependent on the value of the **overwrite** attribute. If `overwrite=true`, the value being set overwrites the existing variable, beginning at the specified index. If `overwrite=false`, the value being set is inserted into the existing variable, beginning at the specified index.

op1 Specifies whether the first operand of a calculated value is a fixed constant or the value of an existing global variable. Valid values are a fixed constant or the name of a global variable.

op1_type
Specifies whether the value of the first operand of a calculated value is set as a fixed constant or from an existing global variable. Valid values are `string` and `variable`.

op1_index
If the source of the value of the first operand of a calculated value is an indexed global variable, op1_index specifies the number of the index used as the value for the calculation.

op1_shared
If the value of op1 is a global variable, shared specifies whether this global variable is shared between applications running in the same rich client environment.

op Specifies the type of operation to occur between the first and second operands of a calculated value. Valid values are concatenate, + (add), - (subtract), * (multiply), / (divide), and % (modulo).

op2 Specifies whether the second operand of a calculated value is a fixed constant or the value of an existing global variable. Valid values are a fixed constant or the name of a global variable.

op2_type
Specifies whether the value of the second operand of a calculated value is set as a fixed constant or from an existing global variable. Valid values are `string` and `variable`.

op2_index
If the source of the value of the second operand of a calculated value is an indexed global variable, op2_index specifies the number of the index used as the value for the calculation.

op2_shared
If the value of op2 is a global variable, shared specifies whether this global variable is shared between applications running in the same rich client environment.

dec Specifies the number of decimal places to which a calculated value is rounded. Valid values are 0–999.

<execute> tag

Defines the action for executing business logic. The attributes of the <execute> tag are:

enabled
Indicates whether this action is enabled for use. The default is `true`.

class Names the Java class that contains your business logic. The class value is required.

method
Names the method inside the class that executes the business logic. The method value is required.

package
Names the package that the Java class resides in on your file system. The package value is optional.

<show> tag

Defines the action for showing a URL. The <show> tag has the following attributes:

enabled
Indicates whether this action is enabled for use. The default is `true`.

template
Specifies the template to be used for this action.

url Identifies the Uniform Resource Locator (URL) of the Web page to show. This attribute is required.

<forwardtoURL> tag

Defines the action for passing control from a project to a JSP that invokes an Integration Object. The <forwardtoURL> tag has the following attributes:

enabled
Indicates whether this action is enabled for use. The default is `true`.

startStateLabel
If forwarding control to a JSP with an Integration Object chain, specifies the start state label of the first Integration Object in the chain to be executed.

url Specifies the URL of the Integration Object JSP.

<disconnect> tag

Disconnects the default connection. Use this action carefully and only for events from which you cannot recover. The <disconnect> tag has the following attribute:

enabled
Indicates whether this action is enabled for use. The default is `true`.

<play> tag

Defines the action for playing a macro. The <play> tag has the following attributes:

enabled
Indicates whether this action is enabled for use. The default is `true`.

macro Names the macro to be played. This attribute is required.

<perform> tag

Defines the action for playing a macro on any connection, not necessarily the default connection. This action does not affect the current host screen. The <perform> tag has the following attributes:

connection
The connection on which the macro is to be played. The default is `main`.

enabled
Indicates whether this action is enabled for use. The default is `true`.

macro The name of the macro to be played.

<pause> tag

Defines the action for waiting for some time before continuing with normal processing. The <pause> tag has the following attributes:

enabled
Indicates whether this action is enabled for use. The default is `true`.

time Specifies the time, in milliseconds, to pause before continuing with normal processing.

<sendkey> tag

Defines the action for sending a specified key to the host screen to perform an action. The <sendkey> tag has the following attributes:

enabled
Indicates whether this action is enabled for use. The default is `true`.

key Indicates key to send to the host screen.

row Defines the starting row on the host screen where the key is to be inserted.

col Defines the starting column on the host screen where the key is to be inserted.

<globalRules> tag

The <globalRules> tag is the enclosing tag for any global rules you define for screen events. It has no attributes.

<rule> tag

The <rule> tag defines a global rule.

The attributes of the <rule> tag for screen customization-level rules are the same as for project-level global rules. However, when you create a screen customization-level and a project-level global rule using the same input field, the screen customization-level rule will have a higher priority. The <rule> tag attributes are:

associatedScreen
The name of a screen capture in the project, from which the global rule is defined.

componentSettings
Any settings configured for the global rule, such as recognition criteria.

description
The description entered when the global rule was created.

enabled
Indicates whether this global rule is enabled. Reflects the state of the check box on the Rendering page of Project Settings.

endCol
The last column of the host screen to which this global rule should be applied. -1 means the rightmost column of the host screen.

endRow
The last row of the host screen to which this global rule should be applied. -1 means the bottom row of the host screen.

name The name that will be shown in the list of global rules on the Rendering page of Project Settings.

startCol
The first column of the host screen to which this global rule should be applied.

startRow
The first row of the host screen to which this global rule should be applied.

transformationFragment
The name of the transformation fragment file associated with this global rule. This file contains the information specifying how to transform the host component. It will be included in a transformation if the appropriate input fields are present in the host screen.

type The pattern type component for this global rule, taken from the first page of the Create Global Rule wizard. The type can be one of the following:

com.ibm.hats.transform.components.InputFieldByTextPatternComponent
This pattern component recognizes input fields on the host screen based on text near the fields.

com.ibm.hats.transform.components.AllInputFieldsPatternComponent
This pattern component recognizes all input fields on the host screen.

com.ibm.hats.transform.components.InputFieldBySizePatternComponent
This pattern component recognizes input fields on the host screen based on the size of the input fields.

The following tags are also included in each specific global rule:

componentSettings
The <componentSettings> tag is the enclosing tag for any settings defined for the pattern type component specified on the type attribute of the <rule> tag. This tag has no attributes.

setting
The <setting> tag is the enclosing tag for any settings defined for the pattern type component specified on the type attribute of the <rule> tag.

The attributes of the <setting> tag are:

name Specifies the name of a customizable setting for the pattern type component. The available settings depend on the component.

- For the com.ibm.hats.transform.components.InputFieldByTextPattern Component, the settings for the name attribute are:

caseSensitive
Specifies whether the case of the text on the text setting must match before the pattern is recognized. Valid values are `true` and `false`. The default is `true`.

immediatelyNextTo
Specifies which input fields you want to transform.Valid values are:

true Specifies that only the nearest input field should be transformed.

false Specifies that all input fields should be transformed.

The default is `false`.

location
Specifies where text in a protected field, as specified on the text setting, must be in relation to input fields for this global rule to be applied. Valid values are:

ABOVE
Specifies that the text must be above the input field.

BELOW
Specifies that the text must be below the input field.

LEFT Specifies that the text must be to the left of the input field.

RIGHT
Specifies that the text must be to the right of the input field.

The default is RIGHT.

text Specifies some text in a protected field of the host screen. Valid values are any text in a protected field on the host screen.

- For the com.ibm.hats.transform.components.AllInputFieldsPattern Component, there are no component settings.
- For the com.ibm.hats.transform.components. InputFieldBySizePattern Component, the setting for the name attribute is fieldSize.Valid values are the sizes of any input fields on the host screen.

<associatedScreens> tag

The <associatedScreens> tag encompasses the screen tag that follows.

<screen> tag

Defines a screen associated with the screen customization. The <screen> tag has the following attribute:

name Specifies the name of a captured screen, for which the screen recognition criteria and actions have been defined.

<description> tag

The <description> tag is the enclosing tag for the description associated with the screen customization, which consists of the <oia> tag and the <string> tag. There are no attributes for the description tag.

<oia> tag

The <oia> tag in the screen customization .evnt file specifies an operator information area (OIA) condition to match. This tag is optional. The default is to wait for inhibit status. The attributes of the <oia> tag are:

status If `NOTINHIBITED`, the OIA must be uninhibited for a match to occur. If `DONTCARE`, the OIA inhibit status is ignored. This has the same effect as not specifying OIA at all. Valid values are NOTINHIBITED and `DONTCARE`. This is a required attribute.

optional
If `false`, this descriptor is considered non-optional during screen recognition. If the descriptors contain more than one non-optional descriptor, and more than one optional descriptor, the non-optional descriptors are checked first. If all of the non-optional descriptors match, the screen matches. If at least one of the non-optional descriptors does not match, the optional descriptors are checked. One of the optional descriptors must match for the screen to match. Otherwise, the screen fails to match. The value must be `true` or `false`. This attribute is optional. The default is `false`.

invertmatch
If `true`, recognition matching passes only if the screen does not match this description element (boolean not operation). The value must be `true` or `false`. This attribute is optional. The default is `false`.

<string> tag

The <string> tag describes the screen based on a string. The attributes of the <string> tag are:

value The string value. This value can contain any valid Unicode character. This is a required attribute.

row The starting row position for a string at an absolute position or in a rectangle. The value must be a number or an expression that evaluates to a number. This value is optional. If not specified, Macro logic searches the entire screen for the string. If specified, col position is required. <erow> and <ecol> attributes can also be specified to specify a string in a rectangular area.

Note: Negative values are valid and are used to indicate relative position for the bottom of the screen (for example, -1 is the last row).

col The starting column position for the string at an absolute position or in a rectangle. The value must be a number or an expression that evaluates to a number. This attribute is optional.

erow The ending row position for string in a rectangle. The value must be a number or an expression that evaluates to a number. This attribute is optional. If both **erow** and **ecol** are specified, string is in a rectangle.

ecol The ending column position for string in a rectangle. The value must be a

number or an expression that evaluates to a number. This attribute is optional. If both **erow** and **ecol** are specified, string is in a rectangle.

casesense
If `true`, string comparison is case sensitive. The value must be `true` or `false` or an expression that evaluates to true or false. This attribute is optional. The default is `false`.

optional
If `false`, this descriptor is considered non-optional during screen recognition. If the descriptors contain more than one non-optional descriptor, and more than one optional descriptor, the non-optional descriptors are checked first. If all of the non-optional descriptors match, the screen matches. If at least one of the non-optional descriptors does not match, the optional descriptors are checked. One of the optional descriptors must match for the screen to match. Otherwise, the screen fails to match. The value must be `true` or `false` or an expression that evaluates to true or false. This attribute is optional. The default is `false`.

invertmatch
If true, recognition matching passes only if the screen does not match this description element (boolean not operation). The value must be true or false or an expression that evaluates to true or false. This attribute is optional. The default is false.

<nextEvents> tag

The <nextEvents> tag encompasses the <event> tag that follows. The <nextEvents> tag has the following attribute:

defaultEvent
Specifies the default screen customization (event) used as the next screen to occur, if there are no matching screen customizations named on the event tags. If defaultEvent does not specify an event, the normal event priority list in the project settings is used. Valid values are:

- `unmatchedScreen`
- `error`
- `disconnect`
- `stop`
- (no value)

<event> tag

Defines another screen customization in the project that is the probable next screen to occur. The <event> tag has the following attributes:

enabled
Indicates whether the screen customization (event) defined on the name attribute is enabled for use. The default is true.

name Specifies the name of a screen customization that is the probable next screen to occur.

<remove> tag

The <remove> tag removes global variables previously added to the screen customization (event). The <remove> tag has the following attributes:

enabled
Indicates whether the global variable defined on the name attribute is enabled for removal. The default is true.

name Specifies the name of the global variable to be removed.

remove Type
Specifies the type of the global variable to be removed. Types include oneLocal, oneShared, allLocal, allShared, and all.

Macro files (.hma)

Macro files are stored in the *project_name*/**profiles/macros** directory. You can view and edit the source of the macro files by double-clicking on the name of the macro in the **HATS Project View** to open the macro editor. The source for the file can be viewed by clicking on the **Source** tab. For more information about macros, see Advanced Macro Guide.

Macro files contain tags that define a set of screens. The tags are described in the sections that follow.

<macro> tag

Begins the definition of the macro. The macro tag has no attributes.

<associatedConnections> tag

The <associatedConnections> tag encompasses the <connection> tag that follows. The attribute of the <associatedConnections> tag is:

default
Identifies the default connection for this macro.

<connection> tag

The <connection> tag identifies the connection with which this macro is associated. The attribute of the connection tag is:

name Identifies the name of the connection with which this macro is associated.

<extracts> tag

The <extracts> tag encompasses the extract tag that follows. The <extracts> tag has no attributes.

<extract> tag

The <extract> tag defines the extraction to occur. The attributes of the extract tag are:

name Specifies the name of the extraction.

handler
You can select a .jsp file to display the extracted information to the end user. A default macro handler is shipped with HATS, and it is named default.jsp. You can find this file by clicking the **HATS Project View** tab of the HATS Toolkit and expanding the project name, and then expanding **Macros > Macro Event Handlers**. If you want to create your own handler, ensure that you return control to the HATS runtime.

showHandler
Specifies whether the extracted information should be shown to the end user. Valid values are true and false.

shared
Specifies whether a global variable being extracted is shared between applications running in the same rich client environment.

save Specifies whether the extracted information is saved to a global variable. Valid values are `true` and `false`.

variableName
If the extracted information is being saved to a global variable, variableName specifies the name of a new or existing global variable.

overwrite
If the extracted information is being saved to an existing global variable, overwrite specifies whether the extracted information is to overwrite the current value of the existing global variable, or whether the extracted information is to be appended to the current value. Valid values are `true` and `false`. `True` specifies that the value of the existing global variable is overwritten.

index If the value being extracted is being written to an existing indexed global variable, this attribute specifies the index number to which the value being set is written. The effect of this attribute is dependent on the value of the **overwrite** attribute. If overwrite=true, the value being extracted overwrites the existing variable, beginning at the specified index. If `overwrite=false`, the value being extracted is inserted into the existing variable, beginning at the specified index.

indexed
Specifies whether the extracted information is a single string or a list of strings. Valid values are `true` and `false`. `True` specifies that the extracted information is a list of strings.

isBidi Specifies whether the connection used in recording the macro is bidirectional. Valid values are `true` and `false`.

isRtlScreen
Specifies whether the bidirectional screen is right-to-left. Valid values are `true` and `false`.

screenorientation
Specifies the orientation of the extract action. Valid values are `ltr` and `rtl`.

<prompts> tag

The <prompts> tag encompasses the prompt tag that follows. The prompts tag has no attributes.

<prompt> tag

The <prompt> tag defines the prompt to occur. The attributes of the <prompt> tag are:

name Specifies the name of the prompt.

handler
You can select a .jsp file to prompt the end user for the necessary information, and include a button for the user to submit the information. A default macro handler is shipped with HATS, and it is named default.jsp.

You can find this file by clicking the **HATS Project View** tab of the HATS Toolkit and expanding the project name, and expanding **Macros > Macro Event Handlers**. If you want to create your own handler, ensure that you return control to the HATS runtime.

source Specifies whether the value of the prompt is set to a string or the value of a global variable. Valid values are `string` and `variable`.

variableName
If the value of the prompt is being saved to a global variable, variableName specifies the name of a new or existing global variable.

variableIndex
If the value of the prompt is being saved to an indexed global variable, variableIndex specifies to which index the value should be assigned. This value is always `0`.

variableIndexed
Specifies whether the information for the prompt is coming from an indexed global variable. Valid values are `true` and `false`. `True` specifies that the global variable is indexed.

value Specifies either the string to be used for the prompt or the name of a global variable from which the value is taken.

welApplID
Specifies the application ID to use with the WEL logon macro.

welIsPassword
Specifies whether this is a password field to use with the WEL logon macro.

LTRImpicitOrient
Specifies whether the implicit bidirectional screen orientation is left-to-right. Valid values are `true` and `false`.

isBidi Specifies whether the connection used in recording the macro is bidirectional. Valid values are `true` and `false`.

isRtlField
Specifies whether the bidirectional field is right-to-left. Valid values are `true` and `false`.

isRtlScreen
Specifies whether the bidirectional screen is right-to-left. Valid values are `true` and `false`.

screenorientation
Specifies the orientation of the prompt action. Valid values are `ltr` and `rtl`.

<HAScript> tag

The <HAScript> tag is the main enclosing tag for the other macro tags and attributes. See the *HATS Advanced Macro Guide* for more information about macro tags.

Screen capture files (.hsc)

Screen capture files are XML representations of host screens, used to create or customize screen customizations, screen combinations, transformations, global rules, or macros.

Screen capture files are stored in the *project_name*/Screen Captures directory. You can view these files by double-clicking on the name of the screen capture in the **HATS Project View**. You cannot edit screen capture files.

Note: Screen captures of video terminal (VT) host screens can be used to create or customize a macro using the Visual Macro Editor and as the check-in screen when configuring pooling. They cannot be used to create screen customizations, screen combinations, transformations, default rendering, or global rules.

BMS Map files (.bms and .bmc)

Basic Mapping Support (BMS) maps are screen definitions files for Customer Information Control System (CICS®). Each map defines all or part of a screen, and a CICS application typically displays one or more maps to create a complete screen image. The source for BMS maps is organized in groups called map sets. One map set contains one or more maps. Map sets exist in source form as one map set per source file.

BMS map set files can be imported into a project in HATS Toolkit. When HATS imports BMS maps, the import takes place at the map set level. It is not possible to import an individual map. Imported BMS map set files have a file extension of .bms, and the individual maps have a file extension of .bmc in HATS Toolkit.

Both the map set files (.bms) and the map files (.bmc) are stored in a separate **Maps** folder within the HATS project. By default, the **Maps** folder is not visible in the **HATS Project View** until there are maps imported.

HATS enables you to generate screen captures from map files. You can choose to generate the screen captures when you import map sets, or you can generate them from the maps after they are in the **Maps** folder. To generate screen captures from maps, right click on a map file to display the pop-up menu and select **Generate Screen Captures**. You can elect to create separate screen captures for each BMS map selected or merge selected BMS maps into a single screen capture. Maps cannot be merged if fields overlap. Once the screen captures are created, you can begin creating HATS screen customizations.

You can open a map set file in a HATS Toolkit editor by double-clicking on the file in the **HATS Project View**. See the *CICS Application Programming Reference* for information about the contents of the file. When a map set file is modified and saved in the text editor, the maps that make up the file are regenerated, with one exception: map files in which the contents of fields defined with labels in the map set files have been modified. To regenerate those maps, you must import the source file again using the BMS map set import wizard.

When a CICS application runs, it can modify the contents of the fields defined with labels. You might need to create screen captures with the fields appearing as they will be when the CICS application runs. Since the labeled fields are changeable when the application runs, the map set file (and the map files that are in the map set) may not contain all the information needed to generate an actual screen capture. While you cannot edit map files, double-clicking on a file opens a file preview. The property sheet view in HATS Toolkit enables you to add missing information and manually set the contents of the fields. By modifying the contents of the fields, a single map can be used for multiple screen captures.

Notes:

1. When you are previewing a map file in HATS Toolkit, the fields displayed in the property sheet view are the fields for the map file highlighted in the **HATS Project View**, not the map file you see in the preview window.
2. You can also screen capture files using the property sheet view in HATS Toolkit, as long as the screen capture files were generated from BMS map files.

Image files (.gif, .jpg, or .png)

Image files are used in HATS Toolkit within template files to create the page displayed to the user of your project.

Image files are stored in the *project_name*/**images** directory. You can view the image files by double-clicking on the name of the image.

Spreadsheet files (.csv or .xls)

Spreadsheet files in either .csv (comma separated values) or .xls (Microsoft Excel) format can be automatically generated from host screen data defined within the TableWidget. The spreadsheet files are created by the HATS SpreadsheetGeneratorServlet and can be displayed at runtime when the user clicks a defined button or link. A dialog popup displays, and the user can type the directory and file name where the spreadsheet files are to be stored.

For information about creating spreadsheet files, see the Table widget in the *HATS User's and Administrator's Guide*.

Host simulation trace files (.hhs)

Host simulation trace files can be saved and then used to run HATS in a simulated host connection environment instead of using a live host connection. Simulations are created by the Host Simulator Recorder, which acts as a proxy between the real host and the HATS terminal. The host simulation trace files are created in XML format with a file extension of .hhs and are stored in the following directory in the **Host Simulations** folder, which is accessed from the **HATS Projects** view:

- RCP projects - *Project_name*/profiles/hostsimulations

For information about creating host simulation trace files, see the *HATS User's and Administrator's Guide*.

ComponentWidget.xml

The ComponentWidget.xml file contains the definitions of all the host components and widgets provided with HATS. If you add your own host components or widgets, you will need to update this file. For an explanation and a small sample of the file, see "Registering your component or widget" on page 70. The ComponentWidget.xml file appears as the last item in your project in the **Navigator** view. To edit the file, double-click the file name in the **Navigator** view and select the **Source** tab.

For a description of the contents and use of this file, see "Registering your component or widget" on page 70.

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information might include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements or changes in the product(s) and the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
5 Technology Park Drive
Westford, MA 01886
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This Rich Client Platform Application Programmer's Guide contains information on intended programming interfaces that allow the customer to write programs to obtain the services of HATS.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Glossary

action. A defined task that an application performs on a managed object as a result of an event, such as a host screen matching the screen recognition criteria specified for a screen event. A list of actions is part of the definition of each event.

ADB. See **application data buffer**.

administrative console. The HATS administrative console is a Web-based utility that provides views and functions to manage licenses and connections, set log and trace settings, view messages and traces, and perform problem determination for HATS Web applications.

application. See **HATS application**.

application data buffer. The format of data that is returned by the WebFacing Server for consumption by the WebFacing application.

application event. A HATS event that is triggered by state changes in the application's life cycle. Examples of application events include a user first accessing a HATS application (a Start event), or an application encountering an unrecognized screen (an Unmatched Screen event).

application keypad. A set of buttons or links representing HATS application-level functions. (Contrast with **host keypad**.)

artifact. See **resource**

background connection. Any connection defined in a HATS application other than the default connection. HATS does not transform screens from background connections. (Contrast with **default connection**.)

bidirectional (bidi). Pertaining to scripts such as Arabic and Hebrew that generally run from right to left, except for numbers, which run from left to right.

BMS map. A screen definition file used with Basic Mapping Support in CICS. A BMS map defines a set of fields which are to be displayed as a group by a CICS application

business logic. Java code that performs advanced functions, such as interacting with other applications, databases, or other systems accessible via Java APIs. Business logic is invoked as an action in an application or screen event.

checkin screen. The screen identifying the host screen that should be active for a connection to be considered ready to be returned to the connection pool. If the application is not on the screen specified by the checkin screen, the connection will be discarded or recycled in attempt to return the connection to the host screen specified by the checkin screen. The checkin screen is only meaningful if connection pooling is specified for a connection.

component. A visual element of a host screen, such as a command line, function key, or selection list. HATS applications transform host components into widgets.

connection. A set of parameters used by HATS, stored in an .hco file, to connect to a host application. (See also **default connection** and **background connection**.)

connection pool. A group of host connections that are maintained in an initialized state, ready to be used without having to create and initialize them.

credential mapper. The component of Web Express Logon that handles requests for host credentials, which have been previously authenticated by a network security layer. (See **network security layer**.)

DDS map. Data Description Specification map. These maps define the layout and behavior of the presentation space for IBM i terminal applications.

Debug. For rich client projects, the same as Run, and in addition enables you to:

- Use the display terminal to see host screens as they are navigated while testing your project
- See debug messages in the Rational SDP console

- See changes you make to your project, for example changing the template or a transformation, without having to restart your application
- Modify and test runtime settings, defined in the runtime-debug.properties file, without modifying the settings, defined in the runtime.properties file, that are deployed to the runtime environment
- Step through Java code, such as HATS business logic

Debug on Server. For Web projects, the same as Run on Server, and in addition enables you to:

- Use the display terminal to see host screens as they are navigated while testing your project
- See debug messages in the Rational SDP console
- See changes you make to your project, for example changing the template or a transformation, without having to restart your application on the test server
- Modify and test runtime settings, defined in the runtime-debug.properties file, without modifying the settings, defined in the runtime.properties file, that are deployed to the runtime environment
- Step through Java code, such as HATS business logic

default connection. The connection on which HATS transforms and presents host application screens to the user. Also referred to as **transformation connection**. (Contrast with **background connection**.)

default rendering. The method used by HATS to render parts of the host screen for which no specific transformation is specified.

deploy. To make a HATS application ready for use in a runtime environment. For HATS Web applications, this includes exporting the HATS project as a Java EE application, that is, as an .ear file, and installing it on WebSphere Application Server. For HATS rich client applications, this includes exporting the HATS project as an Eclipse feature and installing it on individual client systems, either as a stand-alone Eclipse application or from an update site to an existing Eclipse runtime environment.

descriptor. See **screen recognition criteria**.

developer. The person who uses HATS Toolkit to develop applications; also application developer or Web developer. (Contrast with **user**.)

Device Runtime Environment (DRE). A package containing other runtime environments, including the J2SE runtime, which is required to run HATS rich client applications in Lotus Expeditor Client V6.2.0 and earlier. The DRE installs into the runtime environment for Lotus Expeditor Client.

display terminal. A terminal window that displays host screens you can use while testing and debugging to observe interactions between a HATS application and a host application at runtime. You can also interact with the host application using host screens in the terminal window.

Eclipse. An open-source initiative that provides ISVs and other tool developers with a standard platform for developing plug-compatible application development tools. Eclipse is available for download from http://www.eclipse.org.

editor. An application that enables a user to modify existing data. In HATS Toolkit, editors are used to customize resources that have been created by wizards.

Enhanced Non-Programmable Terminal User Interface (ENPTUI). Enables an enhanced interface on non-programmable terminals (NPT) and programmable work stations (PWS) over the 5250 full-screen menu-driven interface, taking advantage of 5250 display data stream extensions.

enterprise archive (EAR). A specialized Java archive (JAR) file, defined by the Java EE standard used to deploy Java EE applications to Java EE application servers. An EAR file contains enterprise beans, a deployment descriptor, and Web archive (WAR) files for individual Web applications. (Sun)

Enterprise JavaBeans (EJB). A component architecture defined by Oracle for the development and deployment of object-oriented, distributed, enterprise-level applications. (Oracle)

event. A HATS resource that performs a set of actions based on a certain state being reached. There are two types of HATS events, application events and screen events.

export. To collect the resources of a HATS project, along with the necessary executable code, into an application EAR file (for Web applications) or Eclipse feature (for rich client applications) in preparation for deploying the application.

Extensible Markup Language (XML). A standard metalanguage for defining markup languages that was derived from and is a subset of SGML.

GB18030. GB18030 is a new Chinese character encoding standard. GB18030 has 1.6 million valid byte sequences and encodes characters in sequences of one, two, or four bytes.

global rule. A rule defining how the rendering of specific input fields should be modified based on certain criteria. Global rules are used in customized screens and screens rendered using default rendering. Global rules can be defined at the project level or at the screen event level.

global variable. A variable used to contain information for the use of actions. The values of global variables can be extracted from a host screen or elsewhere, and can be used in templates, transformations, macros, Integration Objects, or business logic. A global variable can be a single value or an array, and it can be shared with other HATS applications sharing the same browser session.

HATS. See **Host Access Transformation Services**.

HATS application. An application that presents a version of a host application to users, either as a Web-enabled application deployed to WebSphere Application Server, a portlet deployed to a WebSphere Portal, or as an Eclipse client-side processing plug-in deployed to an Eclipse rich client platform such as Lotus Notes or Lotus Expeditor Client. A HATS application is created in HATS Toolkit from a HATS project and deployed to the applicable environment. The deployed application might interact with other host or e-business applications to present combined information to a user.

HATS EJB project. A project that contains the HATS EJB and Integration Objects that other applications can use to get host data. A HATS EJB project does not present transformed screens from a host application.

HATS entry servlet. The servlet that is processed when a user starts a HATS Web application in a browser.

HATS project. A collection of HATS resources (also called artifacts), created using HATS Toolkit wizards and customized using HATS Toolkit editors, which can be exported to a HATS application.

HATS Toolkit. The component of HATS that runs on Rational SDP and enables you to work with HATS projects to create HATS applications.

Host Access Transformation Services (HATS). An IBM software set of tools which provides Web-based access to host-based applications and data sources.

host component. See **component**.

host keypad. A set of buttons or links representing functions typically available from a host keyboard, such as function keys or the Enter key. (Contrast with **application keypad**.)

host simulation. Host simulation enables you to record host simulation trace files that can be saved and then used instead of a live host connection. The recorded trace files can be played back to create screen captures, screen events, and screen transformations using the host terminal function, create and test macros using the host terminal function, test HATS applications using the Rational SDP local test environment, and, along with other traces and logs, aid in troubleshooting a failing scenario in a runtime environment.

host simulation trace. Host simulation trace files record host screens and transactions that can be saved and played back later instead of using a live host connection. Trace files can be recorded using the host terminal function or while in the runtime environment.

host terminal. A HATS Toolkit tool. A session tied to a particular HATS connection, which the HATS developer can use to capture screens, create screen customizations, and record macros.

HTML. Hypertext Markup Language.

HTML widget. See **widget**

Integration Object. A Java bean that encapsulates an interaction with a host screen or a series of host screens. Integration Objects are constructed from macros and can be included in traditional (WSDL-based) Web services, RESTful Web services, or HATS EJB projects. Integration Objects cannot be used in rich client platform applications.

interoperability. The ability of a computer or program to work with other computers or programs.

interoperability runtime. Common runtime used by a combined HATS/WebFacing application to provide management of common connection to the backend host. This runtime decides whether data being returned by the WebFacing server should be handled by the HATS or WebFacing part of the application.

Java Platform, Enterprise Edition (Java EE). An environment for developing and deploying enterprise applications, defined by Oracle. The Java EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, Web-based applications. (Oracle)

JavaServer Faces (JSF). A framework for building Web-based user interfaces in Java. Web developers can build applications by placing reusable UI components on a page, connecting the components to an application data source, and wiring client events to server event handlers. (Oracle)

JavaServer Pages (JSP). A server-side scripting technology that enables Java code to be dynamically embedded within Web pages (HTML files) and run when the page is served, returning dynamic content to a client. (Oracle)

JavaServer Pages Standard Tag Library (JSTL). A standard tag library that provides support for common, structural tasks, such as: iteration and conditionals, processing XML documents, internationalization, and database access using the Structured Query Language (SQL). (Oracle)

JSF. See **JavaServer Faces**.

JSP. See **JavaServer Pages**.

JSR 168. The Java Portlet Specification addresses the requirements of aggregation, personalization, presentation, and security for portlets running in a portal environment. Version 1.0 of the Java Portlet Specification, Java Specification Request 168 (JSR 168), defines standards to enable portlet compatibility between portal servers offered by different vendors. See **JSR 286**.

JSR 286. The Java Portlet Specification addresses the requirements of aggregation, personalization, presentation, and security for portlets running in a portal environment. Version 2.0 of the Java Portlet Specification, Java Specification Request 286 (JSR 286), defines standards to extend the capabilities of Version 1.0 (JSR 168) to include coordination between portlets, resource serving, and other advanced features. See **JSR 186**.

JSTL. See **JavaServer Pages Standard Tag Library**.

keyboard support. The ability for a developer to enable a user to use a physical keyboard to interact with the host when the application is running in a Web browser or rich client environment. The developer also decides whether to include a host keypad, an application keypad, or both, in a project. If keypads are included, the developer decides which keys are included and how those keys and the keypad appear in the client interface.

keypad support. The ability for a developer to enable a user to interact with the host as if the physical keys on a keyboard were pressed, or to perform tasks related to the application, such as viewing their print jobs or refreshing the screen. See also **application keypad** and **host keypad**.

linked HATS/WebFacing project. A project created by linking a single HATS Web project with a single WebFacing project for the purpose of creating an enterprise application that includes a HATS Web application interoperating with a WebFacing application and sharing a connection to a 5250 backend host.

Lotus Expeditor Client. A standalone client of the Lotus Expeditor product. It is installed on a user or development machine.

Lotus Notes Client. A standalone client of the Lotus Notes product. It is installed on a user or development machine.

macro. A macro, stored in a .hma file, automates interactions with the host. It can send commands to the host, enter data into entry fields, extract data from the host, and be used to navigate screens on behalf of the user.

Model 1 Web pages. A single JSP that contains the information to be presented to the user, formatting tags that specify how the information is displayed, and logic that controls the order in which pages are displayed. (Contrast with **Struts Web pages**.)

network security layer. Software that is responsible for authenticating users and authorizing them to access network resources, such as IBM Tivoli Access Manager.

Operator Information Area (OIA). OIA is the area at the bottom of the host session screen where session indicators and messages appear. Session indicators show information about the workstation, host system, and connectivity.

perspective. In the Rational SDP workbench, a group of views that show various aspects of the resources in the workbench. The HATS perspective is a collection of views and editors that allow a developer to create, edit, view, and run resources which belong to HATS applications.

pooling. See **connection pool**.

portal. An integrated Web site that dynamically produces a customized list of Web resources, such as links, content, or services, available to a specific user, based on the access permissions for the particular user.

print support. The ability for a developer to specify a printer session to be associated with a host session, and enable the user to view host application print jobs, send them to a printer, or save them to disk. Print support is available only for the default connection

Profile. For rich client projects, the same as Run, and in addition enables you to locate the operations that require the most time, and identify actions that are repeated, to eliminate redundancy. You can use this function for performance analysis, helping you to get a better understanding of your application.

Profile on Server. For Web projects, the same as Run on Server, and in addition enables you to locate the operations that require the most time, and identify actions that are repeated, to eliminate redundancy. You can use this function for performance analysis, helping you to get a better understanding of your application.

project. A collection of HATS resources (also called artifacts) that are created using HATS Toolkit wizards and customized using HATS Toolkit editors. These resources are exported as a HATS application. Types of HATS projects include Web, portlet, EJB, rich client, and for purposes of administering HATS Web (including portlet and EJB) applications, HATS administrative console projects. See **HATS project** or **HATS EJB project**.

Rational Software Delivery Platform (Rational SDP). A family of IBM software products that are based on the Eclipse open-source platform and provide a consistent set of tools for developing e-business applications.

rendering set. A rendering set is configured by creating a prioritized list of rendering items. Each rendering item defines a specific region in which a specified host component is recognized and then rendered using a specified widget.

resource. Any of several data structures included in a HATS project. HATS resources include templates, screen events, transformations, screen captures, connections, and macros. Other Rational SDP plug-ins sometimes call these "artifacts."

RESTful Web service. See **Web service, RESTful**.

rich client. A plug-in designed to run on the Eclipse Rich Client Platform in a client environment, and designed to provide an enhanced user experience by the appearance and behavior native to the platform on which it is deployed.

Run. For rich client projects, a function in Rational SDP that enables you to test your HATS rich client projects in an Eclipse, Lotus Notes, or Lotus Expeditor Client instance. In this mode you can modify and test the runtime settings, defined in the runtime.properties file, that are deployed to the runtime environment. Be aware that any changes made to the runtime settings while testing in this mode are retained and become effective when you deploy the HATS application to a runtime environment.

Run on Server. For Web projects, a function in Rational SDP that enables you to test your HATS Web and portlet projects in a WebSphere Application Server as appropriate. In this mode you can modify and test the runtime settings, defined in the runtime.properties file, that are deployed to the runtime environment. Be aware that any changes made to the runtime settings while testing in this mode are retained and become effective when you deploy the HATS application to a runtime environment.

runtime settings. Log, trace, and problem determination settings defined in the runtime.properties file that are deployed to the runtime environment.

screen capture. An XML representation of a host screen, stored in a .hsc file, used to create or customize a screen customization, screen combination, transformation, global rule, or macro. Screen captures are useful because they enable you to develop a HATS project even when not connected to the host. They are also useful in creating macros which are the core of HATS Integration Object and Web services support.

Screen captures of video terminal (VT) host screens can be used to create or customize a macro using the Visual Macro Editor and as the check-in screen when configuring pooling. They cannot be used to create screen customizations, screen combinations, transformations, default rendering, or global rules.

screen combination. A type of HATS screen event designed to gather output data from consecutive, similar host screens, combine it, and display it in a single output page. The screen combination definition, stored in a .evnt file, includes a set of screen recognition criteria for both the beginning and ending screens to be combined, how to navigate from screen to screen, and the component and widget to use to recognize and render the data gathered from each screen.

screen customization. A type of screen event designed to perform a set of actions when a host screen is recognized. A screen customization definition, stored in a .evnt file, includes a set of criteria for matching host screens, and actions to be taken when a host screen matches these criteria.

screen event. A HATS event that is triggered when a host screen is recognized by matching specific screen recognition criteria. There are two types of screen events, screen customizations and screen combinations.

screen recognition criteria. A set of criteria that HATS uses to match one or more screens. When a host displays a screen, HATS searches to determine whether the current host screen matches any of the screen recognition criteria defined for any screen event in your project. If HATS finds a match, the defined actions for the screen event are performed.

Screen recognition criteria are also used in the process of recording a macro; in this context they are sometimes called **descriptors**.

Secure Sockets Layer (SSL). A security protocol that provides communication privacy. SSL enables client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, and message forgery. SSL was developed by Netscape Communications Corp. and RSA Data Security, Inc.

source. The files containing the markup language that define a HATS project or one of its resources. Also the name of a folder contained in each HATS project.

SSL. See **Secure Sockets Layer**.

standard portlets. Portlets that comply with the standard portlet APIs defined by Java Portlet Specifications JSR 168 or JSR 286. See **JSR 168** and **JSR 286**. .

Standard Widget Toolkit (SWT). An Eclipse toolkit for Java developers that defines a common, portable, user interface API that uses the native widgets of the underlying operating system.

Struts Web pages. Web pages built using the Apache Software Foundation's Struts open-source framework for creating Java web applications. This method of building Web pages creates class files that set values and contain getters and setters, input and output JSPs, and a Web diagram to display the flow and logic of the Web pages. (Contrast with **Model 1 Web pages**.)

SWT. See **Standard Widget Toolkit**.

system screen. An IBM i screen for which data description specification (DDS) display file source members are not available. System screen is specific to an application on an IBM i platform that has been WebFaced.

template. A template, stored in a .jsp file (for Web projects) or a .java file (for rich client projects), controls the basic layout and style, such as color and font, of the application. It also defines the appearance of areas that are common in your GUI, such as a banner and a navigation area.

text replacement. A HATS function used to transform text on a host system into images, HTML code, or other text on a HATS screen transformation,

theme. A theme groups a set of common appearance and behavior characteristics for a project. These attributes can be individually modified later.

transfer. To copy an application EAR file to the server, typically by FTP.

transformation. A transformation stored in a .jsp file (for Web projects) or a .java file (for rich client projects) defines how host components should be extracted and displayed using widgets in your GUI.

transformation connection. See **default connection**.

transformation fragment. A HATS resource that contains the content with which to replace all occurrences of a pattern in any given transformation.

Unicode. A universal character encoding standard that supports the interchange, processing, and display of text that is written in any of the languages of the modern world. It also supports many classical and historical texts in a number of languages. The Unicode standard has a 16-bit international character set defined by ISO 10646.

user. Any person, organization, process, device, program, protocol, or system that uses the services of a computing system.

user list. A list containing information about accounts (user IDs) that a HATS application can use to access a host or database. User lists contain user IDs, passwords, and descriptions for the accounts.

UTF-8. Unicode Transformation Format, 8-bit encoding form, which is designed for ease of use with existing ASCII-based systems.

Web archive (WAR). A compressed file format, defined by the Java EE standard, for storing all the resources required to install and run a Web application in a single file.

Web Express Logon (WEL). A HATS feature that enables users to log onto several hosts using a set of credentials that are authenticated by a network security layer. (See **network security layer**.)

Web service. A self-contained, self-describing modular application that can be published and invoked over a network using standard network protocols.

Web service, RESTful. A Web service that uses a stateless architecture and is viewed as a resource rather than a function call. Well-formatted URIs are used to identify the Web service resource, HTTP method protocols are used to do create, retrieve, update, and delete (CRUD) activities, and HTTP header information is used to define the message format.

Web service, traditional, WSDL-based. A Web service where typically, XML is used to tag data, SOAP is used to transfer data, WSDL is used for describing the services available, and UDDI is used for listing what services are available.

WebFacing feature. The IBM WebFacing Tool for IBM i feature of the HATS Toolkit. The WebFacing feature provides the ability to convert IBM i data description specification (DDS) display file source members into a Web-based user interface for existing 5250 programs.

WebFaced application. A Web application produced by the WebFacing feature of the HATS Toolkit.

WebSphere. An IBM brand name that encompasses tools for developing e-business applications and middleware for running Web applications. Sometimes used as a short name for WebSphere Application Server, which represents the runtime half of the WebSphere family of products.

WebSphere Application Server. Web application server software that runs on a Web server and that can be used to deploy, integrate, run, and manage e-business applications. HATS applications, when exported and transferred to a server, run as WebSphere Application Server applications.

WEL. See **Web Express Logon**.

widget. A reusable user interface component such as a button, scrollbar, control area, or text edit area, that can receive input from the keyboard or mouse and can communicate with an application or with another widget. HATS applications transform host components into widgets.

wizard. An active form of help that guides users through each step of a particular task.

workbench. The user interface and integrated development environment (IDE) in Eclipse-based products such as Rational SDP.

XML. See **Extensible Markup Language**.

Various Java definitions reprinted with permission from Oracle.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Readers' Comments — We'd Like to Hear from You

IBM Host Access Transformation Services
Rich Client Platform Programmer's Guide
Version 9.5

Publication No. SC27-5903-01

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:

- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: 1-800-227-5088 (US and Canada)
- Send your comments via email to: USIB2HPD@VNET.IBM.COM

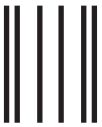
If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC27-5903-01

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Rational Enterprise Modernization UAD
Department 67RA/Building 503
Research Triangle Park, NC 27709-9990

Fold and Tape | **Please do not staple** | Fold and Tape

SC27-5903-01

Cut or Fold
Along Line

Printed in USA

SC27-5903-01