BIRT Chart Library Engine Extensions

org.eclipse.birt.chart.engine Plug-in Extension Specification

This document provides details about the various usage scenarios in which the chart library may need to be extended and covers in depth the chart engine extension points.

Table of Contents

Introduction

A core design feature of the Chart Library is its extensibility. It has been designed in the form of a framework that can support additions and enhancements. To enable this, the framework exposes a few 'Extension Points' that can be used by custom content authors to add their own chart types or chart type components. In addition to providing container services for the custom components, the framework can also be used to access integration-specific services to access services provided by the host environment e.g. an application in which the chart library is being embedded.

References

Chart Library FAQ:

http://www.eclipse.org/birt/

Chart UI Extensions Specification:

http://www.eclipse.org/birt/

Extending the Chart Model:

http://www.eclipse.org/birt/

Various Extensibility Scenarios

This section describes the general scenarios when extension of some or all of the chart library's functionality may be necessary.

Supporting Custom Charts

Although the chart library ships with several chart types, this list does not address the various special requirements the community will have for customized or specialized charting. To this end, a primary extension of the chart library is expected to be custom chart types.

A chart type in the library is represented by an EMF model that maintains the chart structure along with its associated properties. A chart type places some restrictions on associations and containment relationships among its components in a way that provides the most extensible implementation at the same time providing some clear distinguishing characteristics.

At this point, it would help to clarify that the 'chart types' being referred to here are from an end user's perspective. Internally, from the model's perspective, there are two chart types…'**ChartWithAxes**' and '**ChartWithoutAxes**'.

Example:

All the chart types available in the library are internally (their models) special cases of one of the two chart super types **ChartWithAxes** or **ChartWithoutAxes**. Most chart types can be created through a combination of similar special restrictions and custom series / custom rendering.

If however, neither of the super types can support a structure needed for a chart, the chart model schema will need to be modified and the model source re-generated using EMF to include the changes. A discussion on this is outside the scope of this document. This scenario will rarely occur and its use is discouraged.

Supporting Custom Series

A fairly common extension scenario is expected to be custom series types. A series is really the heart of a chart. It holds the actual data to be displayed as well as information about how it should be displayed. All series types implement the Series interface and build on it by providing type-specific properties.

It is possible to create new series types by extending existing series types as is done for scatter series **org.eclipse.birt.chart.model.type.ScatterSeries** but this is not expected to be very common since a series type is expected to be fairly focused in terms of the properties it holds.

A more likely scenario is the creation of a brand new series type (like a *GeographicSeries*). Such an extension will require updating the model to add the new series type and in addition provide the custom model renderer and **DataSetProcessor** extensions. Please refer to the document on extending the chart model for the first part of this task and refer to the sections below for information on creating the engine extensions.

In addition to providing engine extensions, UI extensions can be provided if creation of the model is expected to occur using the Chart Builder provided by the library. Please refer to the UI Extension document for more information.

Example:

The model source in the **org.eclipse.birt.chart.model.type** package illustrates how a custom series can be created. The schema for the individual series types is located in the **xsd/type.xsd** file in the chart engine plug-in with the base series type defined in **xsd/component.xsd**.

Creating Custom Device Renderers

Example: The SwtRendererImpl provides a custom device renderer (dv.SWT) for rendering charts on an SWT canvas. This is provided in the **org.eclipse.birt.chart.device.extension** plug-in.

At times, when a device renderer is created, there is a need to create a custom display server to provide rendering metrics that aid in pre-computing the various primitives off screen. As an example, the SwtDisplayServer provides a custom display server (ds.SWT) that provides SWT metrics to the SWT device renderer.

Creating Custom Model Renderers

Example: The Bar series renderer provides a custom series renderer. This is provided in the **org.eclipse.birt.chart.engine.extension** plug-in. Details on what needs to be implemented are described in the **modelrenderers** extension point section.

Engine Extension Point Usage Details

The Chart Engine plug-in (org.eclipse.birt.chart.engine) exposes the following Extension Points:

- **modelrenderers**
- **displayservers**
- **devicerenderers**
- **datasetprocessors**
- **aggregatefunctions**

These extension points provide a mechanism by which the chart engine can be extended to support custom model content or custom rendering to various output formats (devices).

'modelrenderers' Extension Point

The 'modelrenderers' Extension Point provides a mechanism to add custom series renderers to the chart engine. A series renderer is responsible for drawing graphic elements associated with a series as defined in the chart model.

The Extension writer sets two properties when creating this extension:

'**series**' – The series class (in the model) for which this extension provides rendering services.

'**renderer**' – The actual renderer class which implements the ISeriesRenderer interface and is responsible for rendering the series. Users are expected to subclass the **BaseRenderer** or **AxisRenderer** classes as these provide implementations for all the common elements of the chart for ChartWithoutAxes and ChartWithAxes respectively.

```
// Methods in interface org.eclipse.birt.chart.renderer.ISeriesRenderer

void renderSeries (IPrimitiveRenderer ipr, Plot p, ISeriesRenderingHints isrh) throws
RenderingException;

void renderLegendGraphic (IPrimitiveRenderer ipr, Legend lg, Fill fPaletteEntry, Bounds bo)
throws RenderingException;

void compute (Bounds bo, Plot p, ISeriesRenderingHints isrh) throws GenerationException;
```

Table 1: Interface methods in org.eclipse.birt.chart.render.ISeriesRenderer

The chart engine interacts with the model renderers through the interface methods as follows:

A] Before rendering begins, the generator creates a list of series renders associated with the various chart model series via the extension framework. Each series renderer is notified of a first pass (for any computations needed) via the compute method notification.

B] When rendering begins, the same list of series renders associated with the various chart model series are rendered in the same sequence.

C] For each series renderer, the chart's block layout manager iterates through all of the available blocks and renders them in a loop.

D] If the series renderer happens to be the first in the loop, the rendering sequence walks through all blocks sequenced before the plot block's index. It then draws the plot block's background and then calls renderSeries.

E] If the series renderer happens to be an intermediate one in the loop, the renderSeries method is called.

F] If the series renderer happens to be the last one in the loop, the renderSeries method is called followed by the rendering of all subsequent blocks as defined by the layout manager's block list.

'displayservers' Extension Point

The 'displayservers' Extension Point are intended for implementing a custom display environment that can provide graphics services to the renderers e.g. FontMetrics, display resolution etc.

The extension writer needs to provide the following information:

'name' – The name to be associated with this display server implementation

'server' – The name of the class that implements the **org.eclipse.birt.chart.device.IDisplayServer** interface

```
// Methods in org.eclipse.birt.chart.device.IDisplayServer
void debug ();
void logCreation (Object o);
Object createFont (FontDefinition fd);
Object getColor (ColorDefinition cd);
int getDpiResolution ();
Object loadImage (URL url) throws ImageLoadingException;
Size getSize (Object oImage);
Object getObserver ();
ITextMetrics getTextMetrics (Label la);
Locale getLocale ();
```

Table 2: Interface methods for org.eclipse.birt.chart.device.IDisplayServer

'devicerenderers' Extension Point

The 'devicerenderers' Extension Point has been provided for custom output generation. This extension can be used to provide chart output in a custom format e.g. SVG etc.

The extension needs to provide the following information:

'name' – The name to be associated with this device

'device' – The actual device renderer class. This needs to be an implementation of **org.eclipse.birt.chart.device.IDeviceRenderer**.

```
// Methods in org.eclipse.birt.chart.device.IDeviceRenderer
void setProperty (String sProperty, Object oValue);
Object getGraphicsContext ();
IDisplayServer getDisplayServer ();
boolean needsStructureDefinition ();
void before () throws RenderingException;
void after () throws RenderingException;
```

```
void presentException (Exception ex);
Locale getLocale ();
```

Table 3: Interface methods for org.eclipse.birt.chart.device.IDeviceRenderer

'datasetprocessors' Extension Point

The 'datasetprocessors' Extension Point is provided for creators of custom series types to handle data population of their series instances. Their implementation will be called at runtime to populate the **DataSet** in the series using data obtained by evaluating the **DataDefinitions** associated with the design-time series.

The Extension writer needs to provide the following information:

'series' – The series class (in the model) for which this extension provides data population services

'processor' – The data processor class which implements **org.eclipse.birt.chart.engine.datafeed.IDataSetProcessor** and populates the data for the series

```
// Methods in org.eclipse.birt.chart.engine.datafeed.IDataSetProcessor
DataSet fromString (String sDataSetRepresentation, DataSet ds) throws DataSetException;
String getExpectedStringFormat ();
DataSet populate (Object oResultSetDef, DataSet ds) throws DataSetException;
Object getMinimum (DataSet ds) throws DataSetException;
Object getMaximum (DataSet ds) throws DataSetException;
Locale getLocale ();
```

Table 4: Interface methods for org.eclipse.birt.chart.engine.datafeed.IDataSetProcessor

The DataSetProcessor is used by both the ChartBuilder UI and the Chart Engine. Both access the implementations using the interface methods:

The 'fromString ()' method is called by the ChartBuilder UI to populate the chart model with sample data from the sample data entered by the user. This enables the chart to be correctly displayed in the Preview Window giving the user an idea of what the final chart will look like. The **IDataSetProcessor** is required to provide information if it expects the string containing the sample data representation to be in a particular format. It should provide this information when the UI calls the 'getExpectedStringFormat ()' method. This is called when the user visits the Sample Data page and is shown to the user to ensure that the sample data is entered correctly.

'aggregatefunctions' Extension Point

The 'aggregatefunctions' Extension Point is provided to create custom aggregate functions associated with base series data grouping. These functions are internally invoked to accumulate and aggregate grouped orthogonal data during the chart dataset data binding process.

The Extension writer needs to provide the following information:

'name' – The symbolic name of the function as displayed in the user interface.

'function' – The implementation class that defines the business logic for the aggregate function. The **org.eclipse.birt.chart.aggregate.IAggregateFunction** interface defines various callback methods that are notified during the function processing stage.

```
// Methods in org.eclipse.birt.chart.aggregate.IAggregateFunction
    void accumulate (Object oValue) throws UnexpectedInputException;
    Object getAggregatedValue ();
    void initialize ();
```

Table 5: Interface methods for org.eclipse.birt.chart.aggregate.IAggregateFunction