Eclipse BIRT 1.0.1 Chart Model

Functional Specifications

Draft 1: August 8, 2005

Abstract

This document describes the Eclipse BIRT Chart Model of the 1.0.1 release

Document Revisions

Draft	Date	Primary Author(s)	Description of Changes
1	August 8, 2005	David Michonneau	Initial Version

Contents

1. Introduction

The Chart Model is the hierarchical representation of the visual Chart element. It will be formed from a set of classes, each representing a distinct element from the Chart. The chart structure will be defined using XML Schema files and from this will be generated the EMF model.

2. Structure

2.1 Schema Structure

The model will be defined in the form of an XML Schema spread over multiple files. Each file will contain the definition of types for classes that are to be generated in a single package. The organization of types in the **.xsd** files is as follows:

model.xsd

- Chart
- ChartWithAxes
- ChartWithoutAxes

type.xsd

- BarSeries
- LineSeries
- PieSeries
- ScatterSeries
- StockSeries

attribute.xsd

- FontDefinition
- Fill
- ColorDefinition
- Gradient
- LabelProperties
- LineAttributes

component.xsd

- Axis
- Series
- Labels
- MarkerLine
- MarkerRange
- Grid
- Scale

ChartPreferences

data.xsd

DataSet

DataElement

SeriesGrouping

Trigger

Action

Query

Rule

SampleData

layout.xsd

Block

Plot

Legend

ClientArea

(This is a representative list and is intended to provide an overview of the schema organization. It is not complete)

2.2 Containment Hierarchy

The model will follow a logical containment hierarchy. The general structure of a chart model will be similar. However, in the types of charts provided out-of-the-box by the library, there will be a significant difference based on which two types of charts will be possible:

2.2.1 Hierarchy : Charts With Axes

The difference between these chart types, as the names suggest, will be that one set will have axes while the other will not. Charts with axes include Bar Charts, Line Charts etc. The structure of such charts will be as follows:

The **chart** will contain one or more **axes** (usually 2 or more) in addition to its own properties (like title etc). Each of these axes will then have its own attributes and an associated **series**. The series will usually have an associated **dataset** which holds the actual data to be shown in the chart.

2.2.2 Hierarchy : Charts Without Axes

Charts without axes include Pie Charts, Donut Charts etc. They will have much the same structure as those with axes, except, because they do not have axes, the series will be present directly in the chart itself.

In addition to the structural information, charts hold other information that will be used during data gathering and or rendering. An example of such information would include a string element that holds the scripting-related functions used throughout the chart. The chart will also include information to be used during rendering like the block information for various movable components.

2.3 Structural Extensibility

The schema is designed with extensibility in mind. Almost all the main types have a base type that has only the absolutely essential properties. These can be extended to create fundamentally different types.

The general process of extending types would involve making changes to the schema and regenerating the model source. This would incorporate the new / changed types into the rest of the model and allow for handling them seamlessly.

3. Model Classes

The model classes will be used to describe a chart. They will be generated from the schema files using EMF.

3.1 EMF

The model sources will be generated from the schema files using EMF (http://www.eclipse.org/emf).

"EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications." – **EMF website**

Once the model sources have been generated, the customized code will be developed in an incremental manner. Using EMF will enable us to focus on the core development goals and not have to worry much about the serialization, cloning, updates to the change notification framework, etc. In particular; this will make the model much more extensible because the users extending the model will not have to do anything special to handle model creation, saving and loading as it will be done by through the existing EMF framework.

Some of the reasons for choosing EMF:

Serialization / De-serialization service: This is available in the framework itself. No code needs to be written to save and load the model

Notification framework: Due to the existence of a robust notification mechanism in the framework, we do not need to create one from scratch. This will allow for easier management of undo / redo type functionality.

Consistency of API: Since the API is generated by the framework based on the schema, the form of the API for all parts of the model is consistent. This makes the generated code easier to use for an end user.

Standard form of serialized output: Since the model output is serialized using the framework, the generated output is in a standard (XMI) form used for models.

Ease of extension: Since EMF is an important eclipse project, many eclipse developers are already familiar with it. As such, they can use this knowledge to extend the library a lot easier.

Incremental code: The code generated by EMF is incremental in the sence that changes made to the generated code (or custom code added by hand) is preserved if the model is updated.

Backward compatibility of Models: Most changes to the model do not affect loading of earlier versions of the model. No special handling is needed for this.

3.2 Packaging

The model source is made up of these core packages:

`org.eclipse.birt.chart.model` – This package holds the main chart classes. This will include the Chart class which will be the base class for all chart types and its two main sub-classes 'ChartWithAxes' (representing all charts that have axes) and 'ChartWithoutAxes' representing those without axes.

`org.eclipse.birt.chart.model.attribute` – This package holds the classes representing attributes of the chart and its constituent elements. Classes in this package will include property classes like Bounds, Size etc. as well as more complex and encapsulating classes like FontDefinition and Fill.

`org.eclipse.birt.chart.model.component` – This package holds classes representing components (or parts) that exist in a chart. These would include the Axis and Series classes as well as the more specific Grid and Scale.

`org.eclipse.birt.chart.model.data` – This package holds the 'data'-related classes. This includes DataSet, DataElement and SeriesGrouping.

`org.eclipse.birt.chart.model.layout` – The layout package holds classes that will be used to determine the layout and relative positions of the various main chart components in the chart when it is displayed. This package will include the base class Block and its subclasses Plot, Legend and TextBlock.

`org.eclipse.birt.chart.model.type` – The type package will be used to hold type-specific classes that would usually not be re-usable or further extended. Examples of classes in this package would be BarSeries and LineSeries.

4. General Structure of a Chart

The general structure of a chart is shown in the UML diagram snippet on the next page.

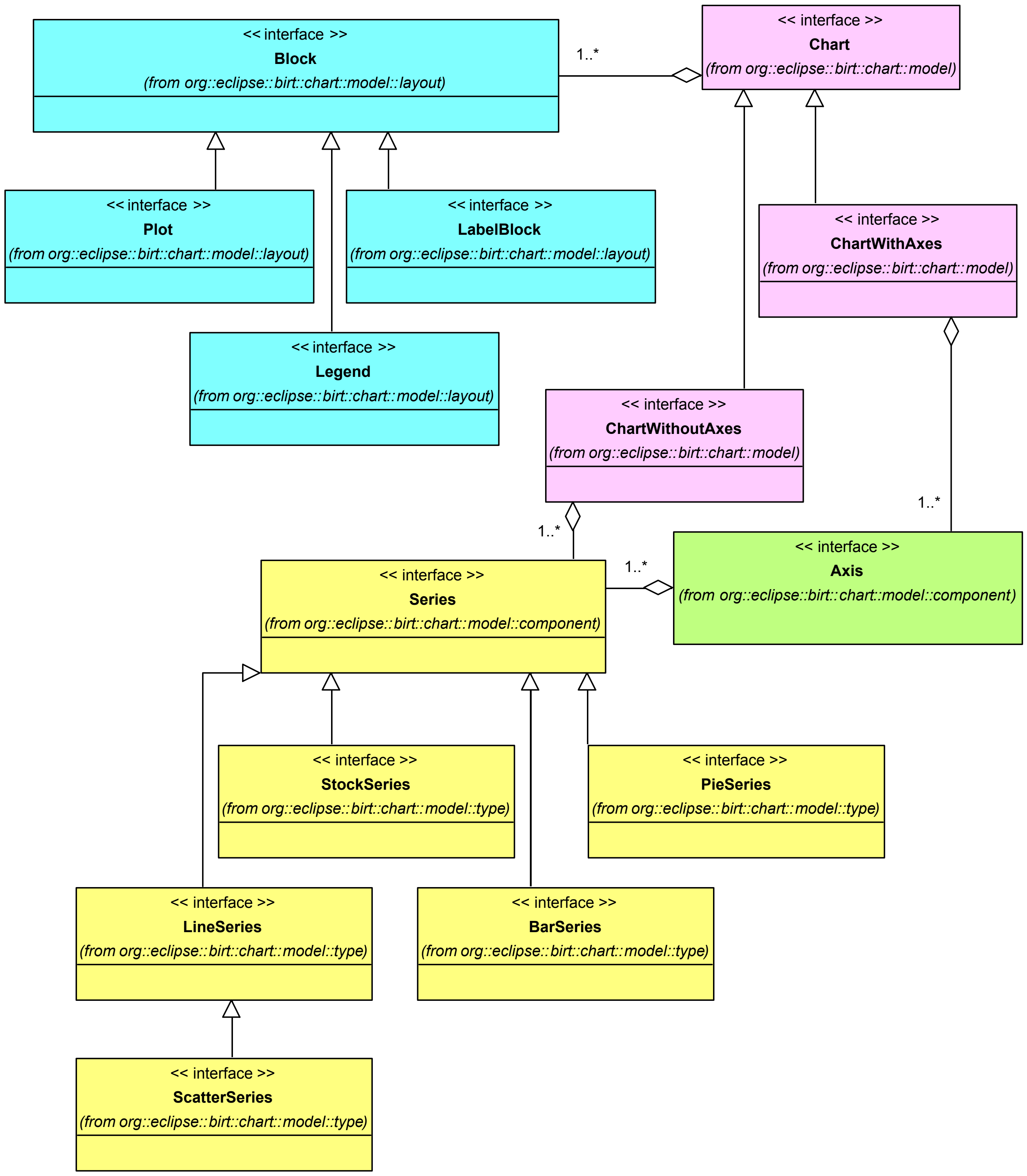
Even though overall, the structure of charts will be similar, there is a major high-level difference based on the type of chart. Most charts that can be supported by the schema would fall under one of two types.

Charts with axes will have one or more axis elements in them. These axis elements will in turn each have one or more associated series. The series will be the ones that hold the actual data (like the category names or data values) as a dataset. In addition to the axes, the chart will also hold layout information in the form of a block. This block holds information about how the chart will be displayed within it (how it will stretch to occupy available space etc.) as well as other blocks that would be contained within it in the display. These would include the 'plot' and 'legend'.

Charts without axes will not have any axes as their name suggests. Instead, the series will be present in the chart itself. The rest of the structure will be the same as that of charts with axes.

Charts without axes provide a more general implementation of the model which isn't tied down to axis definitions.

Interestingly, the extensibility model allows a user to implement a chart with axes and to choose his own rendering algorithm to render the axes.



<< interface >>
Block
(from org::eclipse::birt::chart::model::layout)
<< interface >>
Chart
(from org::eclipse::birt::chart::model)
1..*
<< interface >>
Plot
(from org::eclipse::birt::chart::model::layout)
<< interface >>
LabelBlock
(from org::eclipse::birt::chart::model::layout)
<< interface >>
ChartWithAxes
(from org::eclipse::birt::chart::model)
<< interface >>
Legend
(from org::eclipse::birt::chart::model::layout)
<< interface >>
ChartWithoutAxes
(from org::eclipse::birt::chart::model)
1..*
1..*
<< interface >>
Series
(from org::eclipse::birt::chart::model::component)
1..*
<< interface >>
Axis
(from org::eclipse::birt::chart::model::component)
<< interface >>
StockSeries
(from org::eclipse::birt::chart::model::type)
<< interface >>
PieSeries
(from org::eclipse::birt::chart::model::type)
<< interface >>
LineSeries
(from org::eclipse::birt::chart::model::type)
<< interface >>
BarSeries
(from org::eclipse::birt::chart::model::type)
<< interface >>
ScatterSeries
(from org::eclipse::birt::chart::model::type)

Figure 1. Overview of Chart Structure

5. Using the Chart Model

This section details how the chart model can be used in custom applications. This is very similar to how the model will be created and modified in the Chart Designer UI. The sections below show the code for a simple chart type (**Bar Chart**) but this same process can be extended to use custom types as well.

5.1 Overview

The chart library operates on the chart in the form of a model. This chart model is made up of classes generated from an XSD model schema using EMF. To create a model would generally involve the following steps:

I. Create an instance of the chart (Chart with axes or Chart without axes)

II. Create additional structural components as necessary (Axes, Series etc.)

III. Set the data definitions for various Series / SeriesDefinition elements

IV. Specify the various attributes and values for chart elements

5.2 Create a Chart Instance (with or without axes)

Decide on a chart type to work with (e.g. Bar, Line, Pie, Stock, Scatter or a combination).

If the chart contains axes, use an instance of `ChartWithAxes` else, select `ChartWithoutAxes`. Note that BIRT charts provide pre-defined functionality for rendering a plot with axis in 2D and 2D with depth. If 3D plots are required, this will be supported in a future release.

`ChartWithoutAxes` does not imply that axes may not be rendered on the plot. It presents a more generic implementation of a chart structure that relies on extension implementations to render the entire plot area.

Hence, using the suggested sub-component creation paradigm, we could create a new Bar chart instance as follows:

```
ChartWithAxes cwa = ChartWithAxesImpl.create();
```

This internally creates a chart instance with the following pre-initialized sub-components:

- The various blocks (i.e. Title, Plot, Legend) that comprise the chart
- The title area with a default attributes
- The plot area with default attributes
- Two primary axes (one vertical and one horizontal). Each axis contains:
- Default positioning and font selection of axis labels
- Default caption, positioning and font selection of each axis title
- Default attributes for major and minor ticks
- Intersection of each primary axis against the other axis

- The plot background
- The legend with default block attributes, text attributes and positioning

5.3 Create Chart Components

EMF provides factory classes to create instances of chart components.

e.g. if a new Axis needs to be created, an instance is retrieved using:

`Axis ax = ComponentFactory.eINSTANCE.createAxis();`

This object 'ax' is of type 'AxisImpl' and implicitly cast into an 'Axis' interface reference.

The '`Axis`' interface is located at org.eclipse.birt.chart.model.component

The '`AxisImpl`' class is located at org.eclipse.birt.chart.model.component.impl

'`ComponentFactory`' is located at org.eclipse.birt.chart.model.component.util

Note the color coding used to identify the relationship between the class names, the package names and the method names.

Now, the new '`Axis`' instance created using this method has not been initialized. Default initialization of member variables within a chart component instance (and any other chart API object) is performed using an internal protected 'initialize()' method. Since protected methods are unavailable in public API, appropriate public/static '`create`()' methods have been added in the chart APIs' respective implementing classes for each component.

Hence, a preferred mechanism to obtain a new instance of any EMF object (in the context of the BIRT charting API) could be done using the convenient 'create' method(s) as shown:

`Axis ax = AxisImpl.create([argument1, argument2, …, argumentN]);`

In general, for creation of any 'pre-initialized' chart sub-component, use this mechanism:

`ChartSubComponent csc = ChartSubComponentImpl.create(…)`

To reiterate, default EMF factory classes should not be used to create object instances (unless all appropriate 'setter' methods are called to initialize member variables). Instead, the defined static '`create(…)`' methods should be used. At the time of writing this document, not all `impl` classes contain a '`create(…)`' method. In the absence of a '`create(…)`' method, the EMF factory methods may be used.

5.4 Setting Data Definitions

The `DataSet` classes (`DateTimeDataSet`, `NumberDataSet`, `StockDataSet`, `TextDataSet`) allow to define the data and bind it to the chart. The dataset are bound to the base and orthogonal Series. Here is an example that binds some sample data to a Line Chart:

```
// SAMPLE DATA
Vector vs = new Vector();
vs.add("one");
vs.add("two");
vs.add("three");

ArrayList vn1 = new ArrayList();
vn1.add(new Double(25));
vn1.add(new Double(35));
vn1.add(new Double(-45));

// CREATE THE DATASETS
TextDataSet categoryValues = TextDataSetImpl.create(vs);
NumberDataSet orthoValues1 = NumberDataSetImpl.create(vn1);

// CREATE THE CATEGORY SERIE AND BIND IT
Series seCategory = SeriesImpl.create();
seCategory.setDataSet(categoryValues);

// CREATE THE ORTHOGONAL LINE SERIE AND BIND IT
LineSeries ls = (LineSeries) LineSeriesImpl.create();
ls.setDataSet(orthoValues1);
```

5.5 Setting chart properties

Once the model instance is created, simply use the API to set properties:

Using the API is the same as making any call to a setter method in an ordinary class. Here is an example to set the title to “Hello Chart”

```
ChartWithAxes cwaEmpty = ChartWithAxesImpl.create();
cwaEmpty.getTitle().getLabel().getCaption().setValue("Hello
Chart");
```

6. Extending the Chart Model

The following steps illustrate how a new chart type may be created:

1. Extend the xsd/type.xsd to define your chart type (e.g. `org.eclipse.birt.chart.model.type.CustomSeries`) by referring to existing implementation for provided chart types. Ensure that the ‘**combination**’ property is defined to determine if this custom chart may participate in rendering of a combination chart.
2. Extend xsd/data.xsd to add a custom DataSet (e.g. `org.eclipse.birt.chart.model.data.CustomDataSet`) if needed for the new type.
3. Use EMF’s SDK to regenerate the Java source for all XSDs defined under xsd/

   Ensure that the Java source is generated into a folder that contains the existing source. This ensures that existing manual changes are preserved in the existing Java source and only the new sections are incrementally appended.

   Details on how to generate Java source using XSDs is available at:

   http://download.eclipse.org/tools/emf/scripts/docs.php?doc=tutorials/xlibmod/xlibmod.html

Update the generated Java source into the org.eclipse.birt.chart.engine plug-in project

4. Update your **plugin.xml** to list out the new chart type and its associated class that provides information to the Chart Builder UI; i.e. an implementation of `org.eclipse.birt.chart.ui.swt.extension.IChartType` as an extension value `chartType (name="Custom", classDefinition = "org.eclipse.birt.chart.ui.swt.extension.CustomChart")`
5. Update your **plugin.xml** to add a new property sheet that defines attributes for your custom chart type. i.e. `CustomSeriesAttributeSheetImpl` corresponding to the node in the chart builder UI. i.e. `attributes.series.custom`
6. Write a renderer implementation that subclasses either `AxisRenderer` or `BaseRenderer` in package `org.eclipse.birt.chart.renderer` depending on whether the newly written series could be superimposed as a set of graphic elements in an existing combination chart or if it needs to be rendered standalone in a chart plot of its own. Register this newly written class (e.g. `org.eclipse.birt.chart.render.Custom`) with the corresponding series class that was generated using EMF i.e. `org.eclipse.birt.chart.model.type.CustomSeries,` by defining the modelrenderer extension in your **plugin.xml**