Note: This document is not an implementation plan nor does it necessarily reflect the exact design of Chaco 2. It is still a work in progress and lays out the initial ideas we had.

Chaco 2 Design

Authors: David Morrill, Peter Wang, and Brandon DuRette
Date: 12-Aug-05

1 Introduction

There are a number of existing Python-compatible plotting packages, and they each offer a large variety of (mostly overlapping) standard plot types. Although almost all plot packages allow customization of the appearance of labels, axes, grids, and legends, none of them combine flexible visualization with interactive data manipulation; they are simply not designed with active data manipulation in mind. Also, since none of these packages were built for the purpose of embedding inside other applications, they lack a clean internal component model that demarcates conceptual layers where application writers may extend the toolkit. Without this sort of internal structure, any attempt at extension requires the programmer to understand and interface with all the low-level objects in the system.

The Chaco plotting package has been designed to address these shortcomings. It has two primary requirements: it must be powerful enough to express the myriad kinds of interactive data views needed for the scientific applications we develop, while being simple enough that a scientist can use it from the Python interpreter and visualize data directly with `matlab` or `gnuplot`-like commands.

There is simply no reasonable way to build a monolithic system that meets both of these requirements. Our approach to the problem separates it into two distinct systems: a flexible plotting toolkit with support for rich data and user interaction built into its primitives, and an interactive plotting tool constructed from this toolkit. The toolkit itself consists of separable components with well-defined interfaces to ease the task of extending it for unforeseen plotting

requirements. By adhering to these interfaces, extension writers can develop new plotting primitives that interact well with the rest of the existing components. The interactive tool will assemble specific pieces of the toolkit to expose the most useful set of functionality and plot types to a non-programmer end user.

2 Overview of the New Design

Chaco's primary requirement is to be flexible enough that it can meet the plotting demands of unforeseen scientific applications. This forces the data model to have a mathematical basis, and the presentation layer to be highly customizable and extensible. A treatment of the mathematical rationale for the data model is given in Appendix A.

Chaco solves the problem of visual flexibility by treating all visual components as independent elements on a canvas in any number of configurations. No assumptions are made about the number or location of axes, grids, plot lines, legends, annotations, or any other plotting primitives. Furthermore, user interaction with these elements is treated in an object-oriented way.

All of the objects in Chaco can be broadly divided into three primary categories:

1. The **Data Model** consists of objects representing, transforming, and filtering input data. These objects are generally based on `numarray` objects with thin wrappers to add plotting-related attributes.

2. **Plotting primitives** are the visual components of a plot. Examples are different types of axes, grids, legends, annotations (such as drop lines and pointer arrows), and plot renderers. All plotting primitives are driven by data objects in the Data Model.

3. **Tools** are user interaction objects that can maintain state and can be visualized on the plotting canvas. Though not all tools will need to maintain state or have a visual representation, a tool is fundamentally driven by user-generated events.

Having these objects allows a great deal of flexibility in plot appearance, but it comes at the cost of increased complexity. To mitigate this, Chaco also provides pre-assembled "plot widgets" representing the most common types of plots a scientist or scientific application developer is likely to encounter. These widgets are built entirely from objects in the categories above, and can be customized and extended easily.

The interactive plotting tool is composed of interpreter-prompt-friendly commands that can interact with these plot widgets in a live fashion, and it provides a rich palette of plot editing, annotation, and inspection tools.

3 The Data Model

The Chaco data model consists of arrays and matrices of some basic data types, containers for those arrays, and a pipeline model to combine and filter the containers. A connected pipeline of raw data, intermediate containers, and filters is wrapped up in a DataSource for the end user. The following sections describe each of these elements in more detail.

3.1 Data Series

Chaco has three fundamental data types: scalar, vector (a pair of scalars), and text. **DataSeries** are arrays or matrices of one of these types (along with optional Nulls):

1. **ScalarArray**: An array of scalars.
   Example: `[-0.5, -0.3, 1.1, 2.3, 2.7]`

2. **VectorArray**: An array of vectors. Useful for representing 2D data where there is no regular sampling or it is inconvenient to separate the components into two 1D arrays.
   Example: `[(0.1, -3.2), (0.6, -1.0), (0.87, 0.3), (1.2, 0.8)]`

3. **ScalarGrid**: A 2D matrix of scalars. This is used as an optimized representation for bitmapped, grayscale image data.
   Example:
   ```
   [[0.8, 1.3, 10.9, 12.7],
    [2.2, 2.7, 6.5, 7.4],
    [1.7, 4.4, 3.6, 3.8] ]
   ```

4. **VectorGrid**: A 2D matrix of vectors. Used for vector maps.
   Example:
   ```
   [  [(1.2, 0.3), (-2.2, 2.3), (-0.4, -0.6)],
      [(3.0, 2.5), (1.5, 1.8), (-0.6, 1.7)] ]
   ```

5. **TextArray**: A linear array of text strings.
   Example: `[ “Trial 5”, “Trial 6”, “Trial 8” ]`

In all cases, values may be `None`. TextSeries are used in rare cases when large numbers of text annotations are needed, such as in some bar or histogram plots. Renderers can generally accept them as a batch of labels for the index data series.

The base class for the concrete DataSeries above is AbstractDataSeries:

```
class AbstractDataSeries ( HasStrictTraits ):
    name          = Str
    data_modified = Event
    index_type    = None
    annotations   = List( Tuple( index_type, Annotation ) )
```

```
    def get_data( self, mask[1] = None ):
        # Return data (possibly filtered using optional mask)

    def get_annotations( self, mask = None ):
        # Return list of annotated points and their annotations with
        # optional mask
```

Mathematically, all DataSeries types can be indexed in some fashion, since they are either arrays or matrices. This natural indexing may not always make conceptual sense[2], but it is a mechanism for addressing specific elements in the DataSeries. There are two major uses for the indices: associating annotations with a particular element, and using bitmasks to select a subset of elements. These are both described in more detail in sections below (see "Annotations" and "Selections").

Scalar data arrays are very common, and there are many optimizations and special algorithms that can operate on them if their elements are in numerical order. Consequently, there is a special flag on scalar arrays to indicate this. (Other data series types cannot be meaningfully ordered.)

3.2 Data Channel

A **DataChannel** is any object that can provide an index DataSeries and zero or more value DataSeries. The dimensionalities of the index and value data series don't have to be the same, but the index series must be the same lengths as its value series.

DataChannels are passed as input to plot renderers, and renderers will only work with DataChannels that supply index and value data series of correct dimensionality. If a user wants to visualize slices of data through a multi-dimensional volume, he has to either manually extract slices of appropriate dimensionality to pass into pre-supplied DataChannels, or he can subclass an existing DataChannel type and override the `index` and `value` traits with functions that extract data on request.

The possible combinations of index and value series types are tabulated below.

[1] The `mask` parameter is used for filtering and is explained in more depth in the DataFilter section below.

[2] E.g. a collection of points in a 2D plane has no meaningful concept of "the first point" or "the Nth point".

Index type	Value type	Plot type	Notes
ScalarArray	None	line graph	
	ScalarArray	XY, Scatter, Polar, bar/column,	
VectorArray	None	curves, contours	also same as *Scalar-Scalar* plots
	ScalarArray	3D curves	gradient contours, relief meshes
	VectorArray	Vectored lines	
ScalarGrid		Image	Planes have intrinsic indices.
VectorGrid		Vector field	Planes have intrinsic indices.

The DataChannel base class is relatively generic, and illustrates that a DataChannel is really just a container for DataSeries. The `filters` trait is used to attach DataFilters to the channel, and the `data_modified` event allows other objects to listen for data changes on this data channel.

```
class AbstractDataChannel( HasStrictTraits ):
    _index    = Instance( AbstractDataSeries )
    _values   = List( AbstractDataSeries )
    _index_range = Instance( AbstractDataRange )
    _value_ranges = List( AbstractDataRange )
    filters  = List( AbstractDataFilter )
    data_modified = Event

    def get_index(self):
        # Returns the data elements in the index range
    def get_value(self, series = 0):
        # Returns _values[series], filtered by the value range
    def get_values(self):
        # Returns a list of value DataSeries, all filtered by the
        # value range
```

Concrete subclasses of AbstractDataChannel exist only to explicitly enforce dimensionality, type, and multiplicity constraints on DataSeries. This makes it easy for a third-party object to treat DataChannels in a generic fashion without having to know a priori what sorts of DataSeries they may have.

```
class ScalarDataChannel( AbstractDataChannel ):
    _index  = Instance( ScalarSeries )
    _values = List( ScalarSeries )
    filters = List( ScalarDataFilter )

class PlanarDataChannel( AbstractDataChannel ):
    _index = None
    _u_range = Tuple( Float, Float )
    _v_range = Tuple( Float, Float )
    def get_index(self):
        # Returns
```

3.3 Data Filter

There are a number of different needs for filtering or selecting sub-regions of a data series. The most obvious is the notion of representing a range in data space; other uses include downsampling and generating secondary data such as contour lines or trend lines. There isn't a single abstract interface for filters, since they can take so many forms. However, data filters can be broken down into three major categories:

1. ArrayFilters operate on linearly-indexed data series and generally maintain little to no state. They produce a bitmask as opposed to returning or extracting actual data from their input DataSeries. This allows for easy composition of ArrayFilters and also allows the result of a filter on an index series to easily be applied to multiple value data series.

2. MatrixFilters are like ArrayFilters, but for planar DataSeries[3]. They also return a masking matrix, for the same reasons as ArrayFilter.

3. DataChannelFilters can take an entire channel as input and generate new dataseries as output. Alternatively, they can masquerade as DataChannels, with their own internal dataseries. These can frequently be viewed as "adapters" that allow DataChannels to interact with otherwise incompatible Renderers.

```
class ArrayFilter( HasStrictTraits ):
    inputData = Trait( None, ScalarSeries, PointSeries )

    def get_data( self ):
        # Returns the actual values that the filter accepted
    def get_mask( self ):
        # Returns a bitmask where 1 indicates a passing data point
```

[3] Because filtering can be an expensive process, it's important to make the distinctions and parameterize filter types based on DataSeries type.

```
class ChannelFilter( HasStrictTraits ):
    inputChannel = Trait( None, AbstractDataChannel )
    index = Property( _get_index, _set_index )
    values = Property( _get_values, _set_values )
```

Some examples of data filters are:

- RangeFilter – The most commonly used and simplest subclass of ArrayFilter. It has an internal setting for minimum and maximum value, and it is created by a Range object and attached to the appropriate DataSeries.

- ChannelSplittingFilter – This ChannelFilter takes a DataChannel with one index and N multiple values and creates N new data channels that each have a single index and a single value.

- ContourFilter – This filter accepts a ScalarPlane as input and returns a bunch of PointSeries representing the contours on the ScalarPlane.

3.4 Data Source

DataSources are higher-level Chaco objects that wrap DataChannels and DataFilters into larger bundles of functionality for the end-user. They can be directly interfaced with Renderers of appropriate type, and they are somewhat specialized for the user's problem domain.

One of the key functions of the DataSource is to hide as much of the pipeline machinery from the user as possible. Another feature of DataSources is that they can accept filters that apply to all of their internal data channels. This is used by Tools to apply selection ranges across multiple renderers and DataChannels.

Some example DataSources are:

- XYDataSource – accepts X versus Y input series in several formats: [index, val1…valN], [index, [val1…valN]], [None, val1…valN]. In the last case, autogenerates an integer index series corresponding to the length of the longest value series. Produces DataChannels compatible with any ScalarSeries – ScalarSeries renderer (including Polar).

- HistogramDataSource – Can be configured with uniform number of bins or a custom list of bin widths. Takes an input list and creates the necessary internal DataSeries to represent a histogram of that list on the provided bin space. If multiple input lists are provided, then produces multiple overlapping histograms. Produces DataChannels compatible with any ScalarSeries – ScalarSeries renderer (including Polar).

- ImageDataSouce – Accepts a matrix or list of lists and creates a single image DataChannel. Can accept a colormap optional argument.

3.5 Range

Range objects are fundamental to the notion of plotting. Each DataSeries inside a DataChannel has a Range attached to it. A single Range object might be shared among several different DataSeries (in the same DataChannel or in different DataChannels). Fundamentally the Range produces a RangeFilter that DataChannels use implicitly to filter their DataSeries. However, each Range can be configured to have a set max/min or to be "autoscaling", in which case they set one or both limits according to the maximal values of the DataSeries to which they are linked.

Note that this does imply that a Range knows about its user DataSeries. To manage this effectively, they attach trait event handlers to receive notification of data changes, and they cache values for large DataSeries. They are also optimized to handle sorted DataSeries in a much more efficient manner.

Ranges can also be annotated, and in fact may possess a different annotation for each *type* of DataSeries (represented as an internal map of `{DataSeriesType: Annotation}`).

3.6 SelectionRange

Selections are not a fundamental data type, but are instead a collection of Ranges. They are created by Tools (or programmatically) and are attached to DataChannels in the same way Ranges are. Renderers using those DataChannels can explicitly query for any attached SelectionRanges and render that data in a different manner. Alternatively, SelectionRanges can be attached to a DataSource or series within a DataSource, and the DataSource will alias it to each of its DataChannels.

Although SelectionRanges are primarily used to indicate user-selected data to a Renderer to highlight differently, they can be used as a general way of highlighting sub-ranges of a main data series, both disjoint and continuous. Since each constituent Range object in the SelectionRange can have an annotation,

4 Plotting Primitives

4.1 Fundamentals

4.1.1 Transform

Although a Transform is not a visual element, it plays a crucial (albeit simple) role in the determining the final appearance of any visual element. It is merely an planar geometric transform consisting of scale, translation, and rotation. This transform takes place at the lowest levels of the drawing code; in the case of Chaco, transforms are set up through Kiva and affect all drawing functions called by the visual element.

```
class Transform(HasStrictTraits):
    scale = Float(1.0)
    translation = Tuple(Float, Float)
    rotation = Float(0.0)
```

4.1.2 Style and StyleSheet

All visual elements have aspects of their appearance that can be modified by the user. Renderers have a selection of symbols and line formats; axes have a variety of tickmark mechanisms and options; grids can be rendered with different line styles in each dimension. Almost all visual elements have meaningful border, color, and background appearance options.

To most generally represent this, objects have a list of different "appearance classes" to which they belong; this information is inferred from their class or type and can be explicitly overridden on a per-object basis. Examples of appearance classes are "LineOverlay", "DataPoint", "DataLine", "DataArea", "Box", "Text", etc. Each appearance class has variety of visual options that can apply meaningfully to any object in it, and a specific set of visual options is referred to as a Style.

For example, the LineOverlayDrawer family includes all Axis, Grid, and TrendLine objects. The visual options applicable to it are line width, line style, and line foreground and background colors. BasicLine is then a Style on the LineOverlayDrawer class that specifies a line width of 2 pixels, a stippled line style, and a color of black on white. Styles may choose not to specify a preference for some options; this allows them to be "stacked" or to "mask" one another.

The goal of a Stylesheet is to standardize the appearance of objects of similar type across a plot. In its simplest form, a Stylesheet is just a mapping of appearance class to Style. This would allow for a plot to make all Text labels have the same appearance. The stylesheet can also use more specific keys than just the appearance class; it may contain mappings from specific types to Styles as well. An object's value for a given option is determined by the most specific style mapping that explicitly sets that option. If no Style explicitly sets the option, it defers to the default, background Stylesheet.

Consider a simple plot with two axes and a grid. The underlying plot-wide Stylesheet specifies a linestyle of "black, solid, 2-pixel" for the LineOverlay class. The PlotWidget has, in response to user preferences, created a new entry in the Stylesheet that maps Axis objects to a new LineStyle specifying gray foreground color and 4 pixel width. When each of the axes renders itself, it checks to see if it has any explicit visual preferences set on it. If not, it then consults its Stylesheet and checks for entries for LinearAxis, Axis, and LineOverlay classes (in that order).

4.1.3 Layout

The problem of visual object layout applies to most graphical applications, and Chaco is no exception. However, the layout demands of a plotting system are somewhat more complicated than those of a typical GUI application. Whereas normal application widgets and sub-panels are regularly aligned either with their containers or with adjacent items, Chaco objects may form a

complex set of alignments with objects outside their parent or be subject to a series of “soft” preferences while being laid out on an open space. An example of the former is aligning data labels and annotations between multiple plots, and an example of the latter is positioning contour and data labels on a plot in an aesthetically pleasing manner.

The core layout system is built on two fundamental concepts: anchors and constraints. An anchor represents an (X, Y) position in screen space, and a constraint is a linear algebraic equation or inequality involving the X and Y positions of anchor points. A constraint equation/inequality may be given weights, so that “required” constraints must be satisfied (or the system returns an error) and “non-required” constraints are satisfied in a priority-sorted order.

Although this small set of concepts is sufficiently powerful to express all the layout conditions necessary for Chaco's plotting, they can be cumbersome to use directly. Thus, Chaco additionally offers a higher-level, box-based layout system. By using this system, not only is the programmer saved the trouble of writing myriad constraint relations by hand, but he is also entirely shielded from dealing with constraint weights.

Each visual element in Chaco defines at least one layout box and optional anchors. Users create the visual layout with more intuitive directives specifying positional and size alignment relative to other boxes or anchors within those boxes. Users can also modify box attributes such as margins and aspect ratio. These attributes and inter-box relationships are automatically translated into a set of constraint equations between anchors, which are then passed into the core constraint solver.

Some visual elements will define two layout boxes, one representing the entire rendering area of the element and one representing just the area into which data-related graphics will be rendered. In this way, objects such as axes and renderers can ensure their data representations are faithfully aligned in screen space while being free to add decorators like arrowheads or fancy borders that extend beyond the data region.

Some visual elements (such as the renderer, axis, and grid) may provide auxiliary functions to create anchor points colocated with features of significance. For all data-related visual elements, the origin sometimes holds particular significance; thus, they might implement a GetOriginAnchor() function (which may return None if the origin is not within the visible data range). Visual elements that have a notion of selection may also present functions representing the bounds of the selection region. Renderers may choose to provide anchors corresponding to suitable locations for data legends or other annotation points. There is no generic mechanism for registering or discovering what special anchors are available on a given visual element; using such anchors is purely the responsibility of the programmer assembling the elements within the Frame.

In order for the layout system to assign sensible values to free constraint variables, visual elements must be able to provide size hints about themselves: minimum height/width, maximum height/width, and whether to try to preserve aspect ratio when scaling. The layout system uses these hints to set free variables and create new constraints to express how to divide up free space.

For information on the implementation details of the layout system, see Section 9.

4.2 Visual Elements

This is the abstract base class for all graphical elements in the Chaco system.

```
class AbstractVisualElement(HasStrictTraits):
    parent = Trait(AbstractVisualElement)
    style = Trait(Style)
    stylesheet = Delegate(parent, “stylesheet”)
    styleclasses = List(Trait(StyleClass))
    transform = Trait(Transform)
```

4.2.1 Axis

In general, an axis is a line or curve representing the span of a data series. In most cases, the axis is aligned with the orientation of the index or value data in a renderer. This is not always the case, however, as skew axes may sometimes be embedded in 2D image plots to indicate contour scales or other dimensions of data. In all conceivable cases, an Axis represents some range of values in a single dimension. To reflect this, the Axis base class requires a single Range as its sole input. Any additional commonalities between axis types must be factored into more specialized base classes parameterized on the shape of the axis.

```
AbstractAxis
    AbstractLinearAxis
        LinearAxis
        LogAxis
        BinningAxis
    AbstractCircularAxis
        CircularAxis
        CircularLogAxis
        CircularBinningAxis
    AbstractSplineAxis
        SplineAxis
        SplineLogAxis

class AbstractAxis(AbstractVisualElement):
    range = Trait(DataRange)
    def ToScreenCoords(list_of_values)
```

4.2.2 Grid

There are too many grid types to be able to meaningfully generalize them all into a few core concepts. However, every grid has a direct relationship with the axes lined up along its sides: in almost all cases, each axis defines the intercepts between itself and transverse grid lines. The shape of those transverse grid lines is contingent on the type of the transverse axis, but the Grid can determine how many lines to draw and ask the axis to calculate where to root them. This interaction between grids and axes allows a few simple, general-case grids to be built, and makes

writing custom “strange” grids easier by allowing them to leverage the algorithms already developed for the corresponding axes.

```
class RectGrid(AbstractVisualElement):
    x_axis = Trait(AbstractLinearAxis)
    y_axis = Trait(AbstractLinearAxis)

class PolarGrid(AbstractVisualElement):
    r_axis = Trait(AbstractLinearAxis)
    theta_axis = Trait(AbstractCircularAxis)
```

A generalized spline grid is possible if the Axis object can generate both a position and a transform in screen space for any given data point. (This may be expensive in some cases, but it works well for others.) Each axis's transform is used to draw simplified renditions of the other axis, and the net effect is a spline grid.

```
class SplineGrid(AbstractVisualElement):
    x_axis = Trait(AbstractSplineAxis)
    y_axis = Trait(AbstractSplineAxis)
```

4.2.3 Annotation

An Annotation is one of a large number of different text or symbolic graphics meaningfully attached to a data point or data range. It can be as simple as a text label on a single data point, or as complicated as a trend line approximating a series of selected data points. Drop lines, indicator arrows, data labels, and text boxes are all examples of annotations.

The Frame is responsible for hooking up the Annotation pipeline. To do so, it will pass a DataChannel and a Renderer to an AnnotationRenderer. The AnnotationRenderer can extract the indices of annotated points from the DataChannel, and it can use the Renderer to map those points from data space into world space. It then performs the task of rendering visual primitives such as polygons, symbols, or lines onto the canvas.

4.2.4 Renderer

This class of visual elements is responsible for the actual drawing of data into screen space. It is also responsible for performing translations from screen coordinates back into data space, either as a location in the U and V ranges, or as an index into the data channels.

```
AbstractRenderer
    RectRenderer
        PointRenderer
        LineRenderer
        BarRenderer
        ColumnRenderer
        AreaRenderer
        FilledCurveRenderer
```

```
        ErrorBarRenderer
        CandlestickRenderer
        DistributionRenderer
        ImageRenderer
        MapRenderer
        VectorMapRenderer
    PolarRenderer
        (polar equivalents of above)
```

Most renderers support multiple value data series for a single index data series, e.g. multiple plot lines in a PlotRenderer or stacked bars in a BarRenderer. Also, some of these renderers are thin wrappers for other renderers. For instance, the Bar renderer just wraps the Column renderer with a rotation transformation. (However, the same is not possible in Polar coordinates.)

4.2.5 Frame

The Frame is the object that ties together a group of `VisualElements` (including other Frames) into a single logical "plot". The Frame's responsibilities are limited primarily to layout and basic event dispatch for its visual elements. In general, it is not responsible for connecting any of the `VisualElements` to their respective data sources. However, in some cases where plots are dynamically generated, the `Frame` may take on some data management roles. A `Frame` is essentially a visual template or skeleton for how an actual plot or multi-plot layout will look. Any dynamic behavior that affects this underlying visual skeleton will need to have some portion of its logic in the `Frame`.

Chaco provides some basic frames for the most common types of plots, but more complex multi-plot layouts whose interactions are more sophisticated than what can be expressed at the `PlotWidget` level will require custom `Frames`.

The following examples of real subclasses will help illustrate the role of the `Frame`.

4.2.5.1 SimplePlotFrame

This frame represents the classic simple, rectilinear plot with two axes, a grid, a rendering area (which may host multiple renderers), and a label across the top. This type of frame does not care what type of renderers it may contain - it can be a `LineRenderer` or an `ImageRenderer`, as long as having two axes makes sense. The axes can be linear or log scale, and the grid can be turned off. These are all revealed as styles of the `Frame`, but internally they map to styles on underlying objects. (Note that while changing an axis from one scale to another appears as a style change to the user, it actually triggers a method on the `PlotWidget` containing this `Frame` that swaps axis instances.)

Multiple overlapping renderers can be used to either overlay data or to present the illusion of a single, more complicated plot. For instance, an `ErrorBarRenderer` might be overlayed with two `LineRenderers` to create the appearance of a single Bounded Error Bar plot. (In such a case, it is the parent `PlotWidget`'s responsibility to connect data objects to the auxiliary `LineRenderers`.)

Examples of plots that can utilize `SimplePlotFrame`: XY Scatter, XY Line, Column, Bar, Area, Stacked Area, Histogram, Candlestick/HLOC Stock, Error Bar/Distribution, 2D Image, Vector, 2D Vector

4.2.5.2 PolarPlotFrame

This frame is a standard polar view with a radial axis and an angular axis. It may also contain standard decorators like a title, legend, and grid. Although the default view shows the entire circle, the `Frame` can also show just a quadrant.

Examples of plot types that can utilize PolarPlotFrame: Polar Scatter, Polar Line, Polar Column/Sector graph, Polar Bar, Polar/Circular Area, Polar Vector, Star/Radar graph, Polar Image

4.2.5.3 MapViewFrame

This `Frame` is specially tailored for maps or GIS renderers. It has no axes but has an instance of `CartographicGrid` (or one of its subclasses that represent different projection types) to draw latitude and longitude curves across the rendering area. The only renderer that can be used with this frame is the `MapRenderer`, and there is a set of special actions for this renderer that take into account the degrees-minutes-seconds coordinate system underlying the map data.

Unlike scientific plots where there is typically a single legend that floats inside the plot area, map views frequently have many legends (scale, location, projection parameters, etc) and even embedded sub-plots. `MapViewFrame` supports any number of these legends and sub-frames to be arranged along the inside border of the plot area (in traditional cartographic style).

4.2.5.4 ImageProfileFrame

Although a contour renderer can be placed inside a SimplePlotFrame, the ImageProfileFrame is a multi-renderer layout that consists of a main plot area, two axes, one thin horizontal plot located beneath the X-axis and spanning the width of the main plot, and one narrow vertical plot located to the left of the Y-axis and spanning the height of the main plot.

This frame enables the creation of an ImageProfilePlot that presents a 2D image plot and automatically draws the profile along slices in X and Y centered at whatever point the user clicks in the main plot area.

5 Tools and Interactivity

In Chaco, plotting is not merely a means of visualizing data, but is also a means of interacting with and editing data. This section describes the Chaco event model, the interaction framework, and the default interaction tools that Chaco will provide.

The key design drivers were:

- Interactions (such as selection) should be visible on all views of the same data.

- Each application is unique, so the framework must be extensible enough to allow application developers to create their own interaction paradigms.
- Many applications make use of the same interaction paradigms. To facilitate rapid development of new applications, the common interaction paradigms should be provided by the framework in a way that is not dependent on the data source.

To satisfy these requirements, Chaco provides some basic event dispatch and methods, and provides a model that gives the tool writer maximal flexibility. This does mean, however, that writing an interactive tool will be more challenging than writing a new Renderer type, since the range of possible tools is open ended, and the event-handling system cannot provide as rich of an API.

5.1 Gestures

To simplify the event handling, raw mouse events are hidden from the programmer and wrapped into events called “gestures”. Instead of handling left-down and left-up separately, the programmer sees those two mouse events as a single “left-click” gesture. If left-down and left-up are combined with intermediary mouse-moves, they become “left-drag-start”, “left-drag”, and “left-drag-complete” gestures. Gestures can be qualified by the state of the modifier keys Shift, Control, and Alt. When the user performs a gesture, each visual element below the gesture will be notified. Visual elements pass these gestures along to their gesture event listeners, called “tools”.

A tool may operate in one of three modes: observer, consumer, or dynamic. An observer tool does not get exclusive gesture notification. A consumer tool gets a chance to handle the event and terminate its propagation to other consumer tools (but not observers). Dynamic tools are a hybrid and may choose to consume or merely observe each event as it comes in.

For instance, a "locator" tool that observes mouse move events and updates a status bar is an observer – it expects to receive all mouse move events regardless of what other tools are attached to the plot. An example of a consumer is a “draw” tool that creates new data points on a plot. Once it receives the left-click event, it creates a new point and disallows any other tools from being able to handle the gesture.

To support the three modes of tool behavior, each gesture will generate up to three events per visual element:

- an “observe” event (e.g. “observe_right_click”) that is passed to all the visual elements and on to to their observer tools
- a “pre” event that propagates from the outermost container (and farthest from the user in Z-order) to the innermost
- the actual event that propagates from the innermost object back down to the parent container

These last two events follow the Enable event propagation model and allow for a parent container to block or modify events passing to its children. If at any time a pre-event or event is

marked as handled, event propagation for the gesture is terminated, and no other tools or visual elements have the opportunity to act on the event.

5.2 Object Model

A Chaco Tool is a high-level object that can be as simple as a "mouse_move" event handler that reports data coordinates to something as complex as a polyline drawer on a 2D contour that simultaneously generates profile views into the volume. In an MVC view of Chaco, all Tools are fundamentally Controllers; they originate events that change model state. At a more concrete level, however, Tools themselves sometimes have model that needs to be rendered onto the screen. In the MVC analysis of Tools, they can frequently be modeled by DataSources, and their views are most appropriately generated by plot renderers. A Tool owns all of the incidental data pipeline objects and visual primitives that it creates, and they are destroyed when the tool ceases to be active.

Chaco plots follow the Photoshop model of having a single active tool at any point in time. This tool may have state, such as the line path with the Line drawing tool. Sometimes the tool may have state that persists across tool changes; consider the behavior of the Select tool when the user switches to the Magnify tool to zoom in and then switches back. The user might not necessarily have an explicit tool palette (as in Photoshop); however, the higher-level plot widget may intercept raw mouse or keyboard events and use those to set the active tool for the plot. (For instance, if the user holds down the Shift key, a Zoom tool is instantiated and activated, and if the user holds down the right mouse button and initiates a drag, a LineDraw tool is instantiated.) To support tool persistence across tool changes, the "active tool" is treated as a stack, and tools that should be persistent just get pushed down the stack instead of getting popped off altogether.

5.2.1 GestureCompiler

Subclasses of `GestureCompiler` turn low-level Enable events into gestures. Usually a high-level `Plot` will instantiate a default `GestureCompiler` and either use it for all event handling or use it as the default handler and override certain Enable event handlers to customize functionality. There will be several different `GestureCompilers` provided with Chaco, each representing varying degrees of user interactivity.

5.2.2 EventManager

The `EventManager` is the central coordinator for plot interactions. It has several responsibilities: making existing plot items available to tools when the tools are created; notifying interested tools when new plot items become available; and composing mouse events into gestures and managing the gesture notification process.

The `EventManager` has a list of its `InteractionGroups` and (as a possible optimization) their bounding boxes on the screen. Since it is the top-level event handling object, the `Plot` should contain it.

5.2.3 InteractionGroup

Whereas typical UI applications have an event handling hierarchy that is implicitly identical to the visual layout hierarchy, Chaco breaks this down into two distinct entities and `InteractionGroups` represent the event handling one. Plotting primitives are aggregated into `InteractionGroups` which represent conceptual zones of interaction with a `Tool`. Although each primitive can be part of their own `InteractionGroup`, they usually delegate membership to their parent.

An `InteractionGroup`'s behavioral coherence is defined by its `Tool` stack. If a user has selected a region in one primitive and performs a "deselect" on another primitive in the same `InteractionGroup`, then the selection in the former disappears. Another important feature of `InteractionGroups` is that objects wishing to ignore interactions from their parents or that wish to behave in "strange" ways can do so simply by setting their `InteractionGroup` to `None` or delegating to that of a peer or unrelated object.

In general, however, objects representing views of the same data or drawn near one another should belong to the same `InteractionGroup`; an interactive application that does not adhere to this will probably be disorienting for the user.

5.2.4 Shadow Tools

Although not an explicit class, Shadow tools are Tools driven by events forwarded from other Tools and not explicitly by the user. This allows a single Tool to act as a composite of multiple tools. In general, this is useful for integrating the behavior of Tools on multi-plot layouts, where different renderers of different types require differing visual representations of the same concept.

5.2.5 Examples

5.2.5.1 Selection

Consider a rectangular Selection tool on a scatter plot that has an accompanying histogram located below the X axis. Both plots share the same DataSource and InteractionGroup. When the select tool operates on the scatter plot, it adds a SelectionRange to the DataSource of the scatter plot, which causes both the scatter and the histogram plots to highlight the appropriate data ranges. The Tool also has a visual appearance of "crawling ants" overlaid on top of the Scatter renderer. When the user selects the Data Modification tool to change a point in the dataset, the Selection tool is pushed lower in the tool stack and its visual representation is switched into the "backgrounded" state. When the user finishes data modification, that tool is popped off the stack and the Selection tool is once more in the foreground. This restores its visual representation, and allows it to consume input gestures.

5.2.5.2 Text and pan

The user is using the Text Annotation tool to type a description into a certain region in the plot. He wants to move the text over a bit, and doing so requires panning the plot to the side. The plot

has been configured such that a right-click-drag performs a Pan. As soon as he begins this gesture, the EventManager instantiates a new Pan tool and pushes it onto the renderer's InteractionGroup. The Text Annotation tool receives event notification that it's been backgrounded and it switches off its cursor (leaving the partially filled text box on the screen). The Pan tool then intercepts all the drag gestures as the user pans through the plot. As soon as he releases the right mouse button, the drag_complete gesture is fired and the EventManager pops the Pan tool off the stack. The Text Annotation tool receives another event notification that it's the foreground tool, and draws a cursor to indicate to the user that it's ready to accept more text.

6 Plots

A Chaco `Plot` is a high-level object that encapsulates many of the objects in the data model, the visual layer, and the event handling system. While there are some built-in `Plots` representing the common plot types, application writers should be able to easily assemble their own custom `Plots`, or customize the built-in ones.

6.1 Anatomy of a simple plot

6.2 More complex plots

7 Who Writes What

There are three primary layers of Chaco:

1. Core Functionality: Data primitives, layout primitives, event handling facilities
2. Low-level primitives: Layout primitives,

8 Extensions and Future Direction

Although most of the objects in Chaco can be subclassed to add new types of plotting functionality, this section deals with extending the *conceptual* space of Chaco's underlying visualization model.

8.1 Floating Annotations

Attaching annotations to arbitrary points in data space is not a difficult problem; however, the question of which approach best fits the rest of the Chaco data model *is* somewhat tricky. After more design work, Chaco should be able to seamlessly support labeled ranges and regions that

"float" in the data space. (As a temporary hack, a Null point can be inserted in the DataSeries and an Annotation can be attached to that.) Most of the issues that makes this a non-obvious problem occur in the context of filtering and downsampling.

8.2 2D Meshes

Chaco's support for image data is strictly limited to regular, evenly-spaced matrices of data (with support for NULL values). The data pipeline makes an assumption that all DataSeries can be parametrized either with an index or pair of indices. However, in some cases the most natural representation of 2D data is as a vertex mesh with scalar or vector values at each vertex. Such meshes do not lend themselves to parameterization by either a single or a pair of indices. The existing data model would not have to be significantly refactored (if at all) to support such vertex meshes, but a lot of classes would have to be built to support notions such as Range and bitmasking on a mesh (if those notions are even valid).

8.3 3-D Plots

Many of the concepts in Chaco are simplifications or two-dimensional realizations of general visualization concepts. One of the most implicit assumptions in the current design is that index data series can be at most two-dimensional. Extending Chaco to support 3D plots requires additional complexity across the board, and raises the question of where Chaco ends and VTK begins. There is a great deal of interesting functionality that resides in the gray middle space between the two packages, however. Just by supporting simple orthographic projects and basic surface and mesh rendering, Chaco would be able to render 3D bar charts and other presentation graphics, as well as gnuplot's splot()-style graphs. There are many interesting 3D plots that do not require the full-blown horsepower of VTK, and extending Chaco to support these seamlessly with 2D plots is an obvious extension.

8.4 Interactive Plot Layout

Chaco's plot layout mechanism is hopefully simple enough to be used by novice programmers or scientist hackers, but powerful enough to express multi-plot layouts. This system can still be improved, however. The Interactive Plot Tool can be extended with features to write any given plot to disk as a stand-alone Chaco plot application. These plot applications can have some pre-defined style of accepting input data channels, either from the command line, standard input, or files on disk. Users can then distribute them to other users to plot their data, or integrate the source code of the main Plot widget into their own application.

This tool is only a layout tool, and has no implicit notion of data pipeline or tool configuration. Setting those up will have to be done outside the layout tool.

9 Appendix A: Mathematical basis

This section covers the mathematical basis for the data model presented in section 2, and is presented as a reference for the developers and extension writers.

Fundamentally, the task of plotting is one of transformation. Typically this transformation also includes data extraction and reduction. In its most distilled form, 2D plotting is the process of mapping some indexed data into shapes, colors, or colored shapes on a two-dimensional canvas. The data used for the index can either be *structured* or *unstructured*; that is, it can be characterized as having some underlying regular structure, or it cannot. It can also be one dimensional (like an array) or two dimensional (like a matrix). Lastly, the value at each position can be of 1 or 2 dimensions. At first glance, this breakdown suggests 8 different fundamental data series.

Shape	Structure	Dimension	Name
array	regular	1	regular sequence `[1,3,5,7,9,11]`
		2	regular point sequence `[(1,3), (3,4), (5,6), (7,7)]`
	unstructured	1	sequence `[0.3, 1.1, 4.0, -3.1]`
		2	point sequence `[(-2,1), (3,5), (1,1)]`
matrix	regular	1	grayscale image
		2	vector map
	unstructured	1	non-uniform image
		2	non-uniform vector map

Some of these data series are useless, inefficient, or redundant:

- Regular array data can be represented in the same way as unstructured array data. For optimization purposes, a flag can be placed to indicate that the array is ordered. (This is only meaningful for the 1-dimensional case.)

- Unstructured matrix data isn't really very meaningful or common in data analysis. In case someone needs to represent such, they can always resort to either using 2-dimensional arrays (if the data is sparse) or using a regular matrix coupled with some transformation function to map back into unstructured space.

The remaining data series types are what Chaco uses.

10 Appendix B: Layout Implementation

10.1 Overview

The constraint solver uses an algorithm called Cassowary from a group at the University of Washington. This algorithm is an adaptation of the simplex method for solving linear programming problems to the domain of solving UI layout systems. The authors of Cassowary have a C++ implementation with an accompanying Python wrapper, and we have obtained permission from them to use and re-distribute this implementation under the BSD license.

The layout system consists of constraint solver classes, which are just Python wrappers for the C++ classes in the Cassowary implementation, and box model classes that use or wrap the constraint solver classes.

10.2 Solver Wrapper Classes

- `Variable`: A variable in the constraint optimization problem
- `Constraint`: Abstract base class representing a constraint for the solver
- `LinearExpression`: Represents a mathematical expression involving linear combinations of variables and scalars
- `LinearEquality`: A subclass of `Constraint` and `LinearExpression` that is interpreted as having a value equal to zero.
- `LinearInequality`: A subclass of `Constraint` and `LinearExpression` that is interpreted as having a value greater than or equal to zero.
- `LayoutSolver`: An instance of the actual solver. `Constraints` are added by passing `LinearEquality` or `LinearInequality` instances to the `AddConstraint`() method.

The `Variable` and `LinearExpression` classes overload the arithmetic operators to allow the user to write algebraic expressions involving them. For example, if x and y are instances of `Variable`, the following produces a `LinearExpression`:

```
3*(x + (y-2))
```

If a `LinearExpression` or `Variable` instance encounters any equality or inequality operator (<, >, =, <=, >=), it produces a `LinearEquality` or `LinearInequality` instance, which can then be handed to the `LayoutSolver` as a constraint:

```
3*x <= ((5+y) – x) / 2
```

Note that `LinearEquality` and `LinearInequality` automatically adjust themselves to be expressed in one of the normalized forms:

```
LinearEquality: expr = 0
LinearInequality: expr >= 0
```

Users wishing for more fine-grained control over layout than what the box model offers (see below) can write their own `Constraint` equations using member `Variables` of `Box` and `Anchor` instances. Note that `Constraint` equations are linear equalities or inequalities involving `Variable` and scalar reals; that is, `Variable` instances cannot multiply one another or appear in the denominator of a fraction.

```
myFrame.AddConstraint(2*boxA.width <= boxB.width)
myFrame.AddConstraint(boxA.width <= 0.6 * boxA.height)
myFrame.AddConstraint(boxA.center = boxB.right + 50)
```

With such flexibility comes the burden of making sure the constraints are sensible. The solver will be unable to solve a self-inconsistent set of required constraints and will produce an error.

10.3 Box Model Classes

10.3.1 LayoutObject

LayoutObject is a base class for anything which can interact with the box model layout system. It has a concept of position in X-Y space, relative location to other objects, and spacing between itself and other objects. It also has fields for specifying how the layout system should treat it upon resizes.

```
AlignTrait = Enum('top', 'bottom', 'left', 'right', 'center')
RelativePositionTrait = Enum('above', 'below', 'leftOf', 'rightOf')

class LayoutObject(HasStrictTraits):
    hCenter = Trait(Variable, LinearExpression)
    vCenter = Trait(Variable, LinearExpression)
    autoScaleH = Bool(True)
    autoScaleV = Bool(True)
    maintainAspectRatio = Bool(False)
        If both autoScale fields are False, then maintainAspectRatio is disregarded.

    def SetSpacing(self, target, space)
        Sets the minimum spacing between this anchor and the target object

    def SetAdjacent(self, relPosition, target, offset=0)
        relPosition is an instance of RelativePositionTrait.  Positions this object next to the
        target in the relative position indicated, plus a (possibly negative) offset
```

```
    def HAlign(self, target, offset=0):
        Horizontally aligns this anchor to another anchor or the center of a box, plus a
        (possibly negative) offset

    def VAlign(self, target, offset=0):
        Vertically aligns this anchor to another anchor or a box

    def Align(self, target):
        Centers this anchor on another LayoutObject's center
```

10.3.2 Anchor

An Anchor represents an (x,y) position[4] in screen space that is subject to various constraints. An anchor can optionally be flagged as being X-only or Y-only, leaving its other coordinate a free variable.

```
EdgeTrait = Enum('top', 'bottom', 'left', 'right')
RelativePositionTrait = Enum('above', 'below', 'leftOf', 'rightOf')

class Anchor(LayoutObject):
    def __init__(self, **traits):
        self.x = self.hCenter
        self.y = self.vCenter
```

10.3.3 Box

A layout Box represents a logical box on the screen that can be positioned and constrained using high-level functions. It consists of a set of `Anchors` with some pre-defined constraints relating them; some of these anchors are actual instances of Anchor objects and others are `LinearExpressions` involving anchors.

```
class Box(LayoutObject):
    upperLeft = Trait(Anchor)
    bottomRight = Trait(Anchor)
    fixedWidth = Bool(False)
    fixedHeight = Bool(False)
    fixedAspectRatio = Bool(False)
    def __init__(self, **traits):
        # Aliases:
```

[4] Since the layout constraint engine has no notion of screen space and is merely a symbolic optimization solver, a second instance of the solver can be used to define constraints in other dimensions or spaces. An example use of this is for laying out subplots or annotations in polar (r,theta) space and manually using this to drive the dimensions of rectilinear layout boxes in screen space. (Unfortunately, due to the linear nature of the constraint solver, it is not possible to connect anchor points with non-linear relationships such as sine and cosine functions.)

```
        self.left = self.upperLeft.x
        self.right = self.bottomRight.x
        self.top = self.upperLeft.y
        self.bottom = self.bottomRight.y
        # Linear expressions:
        self.width = self.right – self.left
        self.height = self.top – self.bottom
        self.hCenter = 0.5 * (self.left + self.right)
        self.vCenter = 0.5 * (self.bottom + self.top)

    def HSizeAlign(self, targetBox, offset=0):
        Contrains this box's horizontal size to be equal to target box's horizontal size, plus
        a (possibly negative)  offset.

    def VSizeAlign(self, targetBox, offset=0):
        Like HSizeAlign, but in the vertical dimension
```

10.3.4 LayoutContext and Frame

The `LayoutContext` manages a set of `Anchors`, `Boxes`, and their constraints. It contains an instance of `LayoutSolver` which performs the actual layout calculations. The Chaco `Frame` object, though a `VisualElement`, subclasses this class and interacts with it to properly sequence `Draw`() calls generated by the event system.

One of `LayoutContext`'s core features, besides passing constraint equations down to the solver, is the creation of weak (non-required) constraints reflecting the preferred distribution of excess screen space beyond the minimum screen space required to render all the objects in the plot. Each `LayoutObject` can specify how it wishes to be scaled or treated upon resizes via its autoScale fields, but information about free space distribution resides solely in the `LayoutContext`. This information is represented as a dictionary of entries `(boxReference : percentage)`, where `percentage` defines the amount of the free space allocated to that box. If the percentages do not total 100, then they are re-calculated to reflect the percentage of the new sum. If no percentage is assigned, then the current area (or minimum area) of the box is used.

The `LayoutContext` actually defines three allocation dictionaries: one for total area, one for height, and one for width. The user is encouraged to use either the area dictionary or the height and width dictionaries, but if all three are used, the height and width dictionaries will be mathematically converted and merged into the area dictionary as an implicit step in the free space allocation process.

When specifying space allocation, it is not necessary to specify all the objects that must be resized. Since free space allocation creates a set of weak constraints, if object A and object B are constrained (via the layout functions) to have the same vertical size, specifying a height allocation on A will cause B's height to change as well.

```
class LayoutContext:
    solver = Trait(LayoutSolver)
    width_distribution = TraitDict(LayoutObject, Float)
    height_distribution = TraitDict(LayoutObject, Float)
    area_distribution = TraitDict(LayoutObject, Float)

    def AddConstraint(self, Constraint):
        Adds a constraint to the solver.  This function separates users of the LayoutContext
        from the implementation details of how it interacts with its solver.
```

10.4 Layout Procedure

Although the constraints specify the relative locations and positions of objects, finalizing the actual positions in screen space requires actual sizes of objects to be known. This is done via the following procedure:

1. The Frame containing all the visual elements queries them for their minimum sizes. It converts these minimum sizes into a set of constraints and does a first-pass solve to generate a trial set of positions for the visual elements. It also calculates the minimum size for the Frame as a whole.

2. If the available canvas size is less than the minimum size needed, then the Frame must render onto a scrolled canvas, and the layout process is complete.

3. If the available canvas size is greater than the minimum size needed to render all the components, the Frame's LayoutContext must create a new set of weak constraints to allocate free space amount the various components. It first generates constraints according to the distribution mappings, and then uses some simple heuristics to create constraints for the remaining free variables.

In most common cases, the Frame is responsible for tying together the anchors for its visual elements. Since visual elements subclass Box or Anchor, and since the Frame subclasses LayoutContext, their interactions will flow naturally in the code.

As an example, consider: most XY plots can be described using a generic SimplePlotFrame whose layout consists of two axes, a grid, a title, and a renderer (or multiple overlapping renderers). Such a frame will align the layout boxes of the grid, the axes, and the renderer(s), and then center the title above all the components:

```
class SimplePlotFrame(PlotFrame):
    <...>
    def Layout(self):
        # align the two axes at the lower left
        x_axis = self.x_axis.dataLayoutBox
        y_axis = self.y_axis.dataLayoutBox
        x_axis.left.HAlign(y_axis.right)
```

```
        y_axis.bottom.VAlign(x_axis.top)

        Create a single layout box to represent the layout area of available to the
        renderer(s), and then set all the renderers to reference it.  This significantly reduces
        the number of constraints that have to be solved.
        rendererBox = LayoutBox()
        rendererBox.HAlign(self.x_axis, 'center')
        rendererBox.HSizeAlign(self.x_axis)
        rendererBox.SetAdjacent('top')
        rendererBox.VAlign(self.y_axis, 'center')
        rendererBox.VSizeAlign(self.y_axis)
        rendererBox.SetAdjacent('right')
        for renderer in self.renderers:
            renderer.dataLayoutBox = rendererBox

        self.grid.dataLayoutBox = rendererBox
        self.title.SetAdjacent('above', rendererBox)

        To horizontally center the title above the entire width of the frame, we have to
        specify a manual constraint:
        self.AddConstraint(self.title.hCenter =
                (y_axis.left + rendererBox.right) / 2)

        Distribute all excess width beyond the minimum size to the renderer layout box, but
        split up the exceess height between the renderer layout box and the title, in
        proportion to their minimum size ratios.
        self.width_distribution = { rendererBox: 100 }
        self.height_distribution = { rendererBox: None,
                self.title: None }
```

More sophisticated Frames representing more advanced plot layouts may choose to create new sub-plots and place them below or vertically to the side of the existing plot. Such alignments are possible and readily expressed using a mix of high-level and low-level layout functions.