**Perspectives developer’s guide**

This guide will teach you how to extend Perspective’s functionality with new data-source loaders and viewers.

Perspectives main concepts

Perspectives employs separation between data and the viewers used to visualize it. Developers should design data source loaders and create viewer classes that take these data sources as input/constructor parameters. At runtime one data source can be visualized by multiple viewers.

Note that developers could design a fully self-sustained viewer type that loads its own data but this is not the preferred implementation model.

Each data-source and viewer class needs to be accompanied by a “factory” class that produces data-source and viewer objects of those kinds. These factory classes will be registered with the Perspectives environments and will be used to create data and viewers upon users’ requests.

Data-sources and viewers do not manage their UI menus directly. For instance, a developer will not add to a data-source or viewer class code that adds buttons, treats events and so on. Instead, the developer will declare properties of different kinds: text, integer, boolean, option. Perspectives will create appropriate controls for each such property (text fields for text properties, checkboxes for Booleans, or dropdown menus for option properties) and display them in a designated slot in the Perspectives UI, every time a user selects the data-source of viewer. When users interact with these controls and change the property values, a function gets called within the data-source or viewer (propertyUpdated). Developers should place code to react to such property changes in that function.

Many default types of properties have already been defined in Perspectives but developers can implement new and more complex property types such a color pickers, table loaders, etc.

In perspectives viewers can be linked to each other. What this means is that certain property changes that occur in one viewer are broadcasted to other viewers. For instance, if a developer declares a public selection property on a viewer (e.g., a list of nodes selected by a user in a network viewer) and links this viewer to another viewer then each time the selection is changed in one of the viewers, the other viewer is informed of this change. Multiple links within Perspectives create viewer networks within which property changes are cascaded.

Defining a new data-source

New data-sources extend the very generic DataSource class. Developers can add properties to the their data-source to allow users to populate the data source with content from a file or to process the data-source’s content. Data sources need to call the setValid function once their data content was loaded. Viewers cannot be created from invalid data-sources!

See below the example of a FloatMatrix data-source. The full example is included with the Perspectives distribution. The implementation allows a user to load a matrix of floats from a text file with values delimited by either tabs, commas, or space, and then normalize the values.

**public** **class** FloatMatrix **extends** DataSource {

**float** [][] data;

**public** FloatMatrix(String name) {

**super**(name);

**try** {

//this property will allow users to select a file

//from their system to load

**this**.addProperty("Source File", **new** File(""));

//creating an option (i.e. combobox, dropdown)

//property. Such a property has a list of possible

//options and a currently selected index which changes

//when users select different items.

Options o = **new** Options();

o.options = **new** String[3];

o.options[0] = "TAB";

o.options[1] = "SPACE";

o.options[2] = "COMMA";

o.selectedIndex = 0;

**this**.addProperty("Delimiter", o);

//a single action property: a button entitled

//Normalize will be displayed. Pressing the button

//should normalize the data matrix

**this**.addProperty("Normalize", **new** Action());

} **catch** (Exception e) {e.printStackTrace();}

}

//this function is called when users update the data-source’s

//properties

**public** <T> **void** propertyUpdated(Property p, T newvalue){

**if** (p.name == "Source File"){

Option delim = (Options)(getProperty("Delimiter").value);

**int** delimIndex = delim.selectedIndex;

String d = "\t";

**if** (delim == 1) d = " ";

**else** **if** (delim == 2) d = ",";

File f = (File)newvalue;

fromFile(f.getAbsolutePath(), d);

//this needs to be called to tell Perspectives that

//this data source is valid as an input to a Viewer

//now that data has been loaded

setValid(true);

}

**if** (p.name == "Normalize")

{}//code to normalize the data

}

**public** **void** fromFile(String filename, String delim{

//code to load tabular data from a file

}

}

Any new data-source needs to be accompanied by a Factory class which instruct Perspectives how to create objects of this type. Data factory classes extend DataSourceFactory and implement create and creatorType.

**public** **class** FloatMatrixFactory **extends** DataSourceFactory{

p**ublic** DataSource create(String name) {

**return** **new** FloatMatrix(name);

}

**public** String creatorType() {

**return** "FloatMatrix";

}

}

Finally, the data source factory is registered with the Perspectives Environment:

**public** **static** **void** main(String[] args) {

Environment e = **new** Environment();

...

e.registerDataSourceFactory(**new** FloatMatrixFactory());

...

}

Users can now create FloatMatrices and load content for them.

**Defining a new viewer**

The base class for viewers is Viewer. However, developers should extend one of the following three classes: Viewer2D, Viewer3D, or ViewerGUI. As their name suggests, the first two provide basic functionality for 2D and 3D visualizations. The last one is meant for constructing GUI heavy units that are not actual visualizations but provide important functionality (e.g., lists of data, query and lookup modules, etc) in visualization systems.

**2D Viewers**

2D Viewers provide basic functionality for 2D visualizations. In particular Perspectives wraps such viewers in a ViewerContainer that provides automatic zooming (right-mouse button down + drag, or +/- keys) and panning (left-mouse button down + drag, or arrow keys). It also forwards several mouse and keyboard events to the Viewer2D. The ViewerContainer also manages buffered rendering automatically and developers should only be concerned with implementing the render method.

Viewer2D also provides a simulate() method that developers can re-implement to create animations or simulation. This method is called in a loop before rendering.

Below is a basic and incomplete example of a Heatmap Visualization. For a more advanced example please see the GraphDrawing example.

**public** **class** Heatmap **extends** Viewer2D{

FloatMatrix matrix;

**public** Heatmap(String name, FloatMatrix m){

**super**(name);

matrix = m;

}

**public** **void** render(Graphics2D g) {

//code to render the FloatMatrix

}

**public** **boolean** mousepressed(**int** x, **int** y, **int** button){

//code to select a cell in the heatmap

**return** **false**;

};

}

Similarly to data sources, new viewer implementations need to be accompanied by a factory class. Factory classes for viewers are slightly more complicated because they need to instruct Perspectives about the type of data-sources that the viewer needs to have passed to their constructor.

**public** **class** HeatmapFactory **extends** ViewerFactory {

//developers need to instruct Perspectives what sort of inputs

//a Viewer can take into its constructor. They do this by

//implementing the requiredData function

**public** Vector<RequiredData> requiredData() {

//this viewer requires 1 FloatMatrix

RequiredData rd = **new** RequiredData();

rd.className = "FloatMatrix";

rd.amount = "1";

//possible values for amount are: integers, “+” (meaning

//one or more), or “\*” (meaning 0 or more)

Vector<RequiredData> v = **new** Vector<RequiredData>();

v.add(rd);

//multiple RequiredData objects can be added to the returned

//vector;

**return** v;

}

**public** String creatorType() {

**return** "Heatmap";

}

**public** Viewer create(String name) {

**if** (**this**.isAllDataPresent())

**return** **new** Heatmap(name, (FloatMatrix)**this**.getData().get(0));

else **return** **null**;

}

}

Finally the new type of viewer should be registered, via its factory, with the Perspectives Environment.

e.registerViewerFactory(**new** HeatmapFactory());

**Advanced Notes:** *for time-consuming renderings Viewer2D allows developers to decide if a Viewer should constantly refresh the drawing (the default behavior) or not. The mechanism in place for this is re-implementing the skipRendering() method which by default returns false. Developers can choose to have this method return true whenever the visualization has not changed.*

*By default mouse dragging achieves either zooming or panning of the entire visualization. On occasion, developers may want mouse dragging to do something within the visualization (drag a graph node from one position to another). This is achieved by returning “True” (the event has been processed) from the mouse released or mouse dragged methods. See the GraphDrawing example.*

**3D Viewers**

Not implemented yet.

**GUI Viewers**

Often time visualization systems need a strong GUI component in support for the actual visualization. Examples of typical GUI modules include advanced querying capabilities, various types of sortable lists, data processing units etc. In such cases developers can build independent Viewers that bundle such functionality and then link them in the environment to the actual visualization modules to achieve integration.

GUI viewers have a very low implantation overhead: developers simply extend ViewerGUI and gain access to the main JPanel of the window. However, they then need to use low level Java GUI programming to populate the Viewer with the desired controls. Code for populating the viewer’s panel should be placed in an override of the init() function rather than the constructor.

**public** **class** TestGUIViewer **extends** ViewerGUI {

**public** TestGUIViewer(String name){

**super**(name);

}

**public** **void** init() {

**try** {

addProperty("Property", "a string");

} **catch** (Exception e) {

e.printStackTrace();

}

//get access to the root JPanel of the viewer

JPanel p = getPanel();

p.setLayout(**new** BoxLayout(p,BoxLayout.*Y\_AXIS*));

JButton b = **new** JButton("a button");

p.add(b);

p.add(**new** JButton("another button"));

}

**public** <T> **void** propertyUpdated(Property p, T newvalue) {

//treat property changes

}

}

Factories should be created and registered for GUI viewers as well.

**Managing properties**

Viewers and data-source implement the PropertyManager interface which allows them to declare and process properties. Properties are displayed in a designated spot in the Perspectives’ GUI every time a data source or a viewer is selected. Users can change properties and developers can provide an implementation for a function that gets called every time a property changes value. Depending on the type of a property (e.g., Integer, Boolean, Options), a different control is displayed in the property slot: spin-boxes for Integers, checkboxes for Booleans, combo-boxes for Options.

Developers can either declare predefined types of Properties or implement new and more complex types of their own.

**Declaring Properties**

As already seen in the FloatMatrix example, properties are most often declared in an object’s constructor and have to be surrounded by try/catch blocks. Properties are declared using the addProperty() function introduced by the PropertyManager interface. addProperty() receives two parameters: the name of the property and an initialization object. The initialization object’s type is used to determine the type of the property, it’s value to initialize the property.

**try** {

**this**.addProperty("Source File", **new** File(""));

//the system deduces that this property will be of a “File”

//type

//creating an option (i.e. combobox, dropdown)

//property. Such a property has a list of possible

//options and a currently selected index which changes

//when users select different items.

Options o = **new** Options();

o.options = **new** String[3];

o.options[0] = "TAB";

o.options[1] = "SPACE";

o.options[2] = "COMMA";

o.selectedIndex = 0;

**this**.addProperty("Delimiter", o);

//a single action property: a button entitled

//Normalize will be displayed

**this**.addProperty("Normalize", **new** Action());

} **catch** (Exception e) {e.printStackTrace();}

**Responding to property changes**

The PropertyManager interface provides a function called propertyUpdated() that gets called whenever a property value changes. Below is FloatMatrix’s example implementation of this function. Notice that values generally need to be cast to their correct types.

**public** <T> **void** propertyUpdated(Property p, T newvalue){

**if** (p.name == "Source File"){

Option delim = (Options)(getProperty("Delimiter").value);

**int** delimIndex = delim.selectedIndex;

String d = "\t";

**if** (delim == 1) d = " ";

**else** **if** (delim == 2) d = ",";

File f = (File)newvalue;

fromFile(f.getAbsolutePath(), d);

//this needs to be called to tell Perspectives that

//this data source is valid as an input to a Viewer

//now that data has been loaded

setValid(true);

}

**if** (p.name == "Normalize")

{}//code to normalize the data

}

**Linking viewers**

PropertyManagers can be part of a PropertyManagerGroup and be linked to each other. What this means is that if a property is changed in one of the PropertyManagers (either by user intervention, or through computation) this property changes is signaled to any other linked PropertyManagers. Viewers implement the PropertyManager class while the Perspectives Environment implements a PropertyManagerGroups. This means that Viewers can be linked together.

The way to link two types of Viewers together is by declaring properties with the same name in both Viewers, and marking those properties as visible (by setting the field with the same name). Next, if a user establishes a link between those two types of viewers using the Perspectives interface, the propertyUpdated() function will be called in both viewers.

Imagine the following scenario. The developer is creating a network visualization as a node link diagram. The developer is also creating a GUI Viewer that contains a list of all nodes in a network and a querying interface for searching nodes by various attributes. The developer declares a “Selected” property in both viewers, indicating sets of nodes that are selected. This property is set either by clicking on nodes in the node-link diagram or through queries in the GUI viewer but the viewers can be linked such that a selection in either view is mirrored into the other, albeit in a different form.

Defining new property types

Developers can implement more complex types of properties. To do this they need to declare a new type itself, provide a GUI factory for it by overloading the PropertyWidget class (providing a function that will create the UI controls and event processing that needs to be displayed for this type of data), and then register the new type with the Perspectives environment. Below is an example for the aggregated Options type.

//define a new type - Options

c**lass** Options {

**public** **int** selectedIndex;

**public** String[] options;

}

//implement a new PropertyWidget for this type

**class** OptionsPropertyWidget **extends** PropertyWidget {

JComboBox control;

//this function needs to be implemented to tell Perspectives

//what sort of controls to display for this type of property

// - in this case a JComboBox preceeded by a label with the

//the name of the property

**public** **void** widgetLayout() {

setLayout(**new** BoxLayout(**this**, BoxLayout.*LINE\_AXIS*));

**this**.add(**new** JLabel(**this**.p.getDisplayName()));

**final** PropertyWidget th = **this**;

control = **new** JComboBox(((Options)th.p.value).options);

control.setMaximumSize(**new** Dimension(200000,20));

//manages changes in the ComboBox: if the user changes the

//value of the combobox, we need to call the propertyUpdated

//of the PropertyWidget base implementation with the new value.

//In the case of Options, the new value is the same list of

//options but a new selected index.

ActionListener listener = **new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

Options o = **new** Options();

o.options = ((Options)th.p.value).options;

o.selectedIndex = control.getSelectedIndex();

th.propertyUpdated(o);

}

};

control.addActionListener(listener);

**this**.add(Box.*createHorizontalStrut*(10));

**this**.add(control);

}

//developers also need to instruct the PropertyWidget how to update

//the property’s value if changed from outside the GUI

**public** <T> **void** setPropertyValue(T newvalue) {

control.setSelectedIndex(((Options)newvalue).selectedIndex);

}

}

Finally register a factory for the new property type with the Perspective Environment:

c = **new** Options().getClass();

e.registerNewType(c, **new** PropertyWidgetFactory() {

PropertyWidget createWidget() {

**return** **new** OptionsPropertyWidget();

}

});