version

2.0

BILKENT UNIVERSITY

i-Vis Information Visualization Research Group

Chisio Editor Programmer's Guide

BILKENT I-VIS RESEARCH GROUP

Chisio Editor Programmer's Guide

Computer Engineering Department • Bilkent University
Ankara 06800, TURKEY
Phone +90 312.290.1401 • Fax +90 312.266.4047

Table of Contents

Overview

In this chapter, we give an overview of the ways in which Chisio Editor can be customized by a programmer.

Chisio Editor is a graph visualization tool for creating, editing and layout of *compound* or *hierarchically structured* graphs. The tool features user-friendly interactive creation and manipulation of compound graphs. In addition, a number of popular layout algorithms are provided via Chisio Layout package [1].

Chisio Editor 2.0 is based on the Eclipse Graph Editing Framework (GEF) version 3.1, written in Java 1.5. The tool Web page including other documentation on Chisio as well as this one follows:

http://www.cs.bilkent.edu.tr/~ivis/chisio.html

Use Cases

Chisio Editor can be used for different purposes. If you would like to simply use Chisio Editor as a generic graph editor, please refer to the *Chisio Editor User's Guide*. Or, if you would like to customize the tool for implementing a new layout algorithm (e.g. an algorithm that you developed and would like to test in an interactive tool). Please refer to the *Chisio Layout Programmer's Guide*. However, if your goal is to customize the graph editor for a specific application (e.g. one that is used to draw UML class diagrams or a tool for visualization of social networks);, then continue reading this manual

Before you start the customization, you might like to read the following documents on Graphical Editing Framework (GEF).

- Main documentaion page for GEF:

  http://www.eclipse.org/gef/reference/articles.html

- An overview about the MVC architecture widely used in GEF:

  http://www-128.ibm.com/developerworks/opensource/library/os-gef/

- An example for displaying a UML Diagram:

  http://www.eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html

GEF and Chisio Editor Model

GEF assumes that you have a model that you would like to display and edit graphically. The controllers bridge the view and model (Figure 1). Each controller is responsible both for mapping the model to its view, and for making changes to the model. The controller also observes the model, and updates the view to reflect changes in the model's state. Controllers are the objects with which the user interacts.

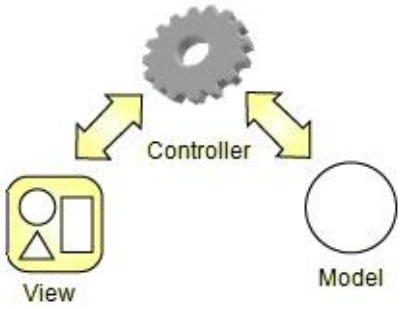

Figure 1 Model-view-controller structure used in GEF [http://www.ibm.com/developerworks/opensource/library/os-gef]

The compound graphs are modeled in Chisio Editor using this framework with the classes shown in Figure 2. A compound node manages a list of children graph objects. A dedicated one is set as the root of the nesting hierarchy. Through recursive use of compound nodes as child objects, an arbitrary level of nesting can be created.

Nodes, edges and compound nodes in Chisio Editor all have distinct properties and UIs, which can be changed by its graphical user interface or through programming. Each node is drawn as a rectangle, ellipse or triangle. Edges can be drawn in a variety of styles. Compound nodes are always drawn with a rectangle, where the name text is displayed on its bottom margin and its geometry is auto-calculated using the geometry of its contents to tightly bind its contents plus user-defined child graph margins. Figure 3 shows the basics of drawing compound graphs in Chisio Editor.

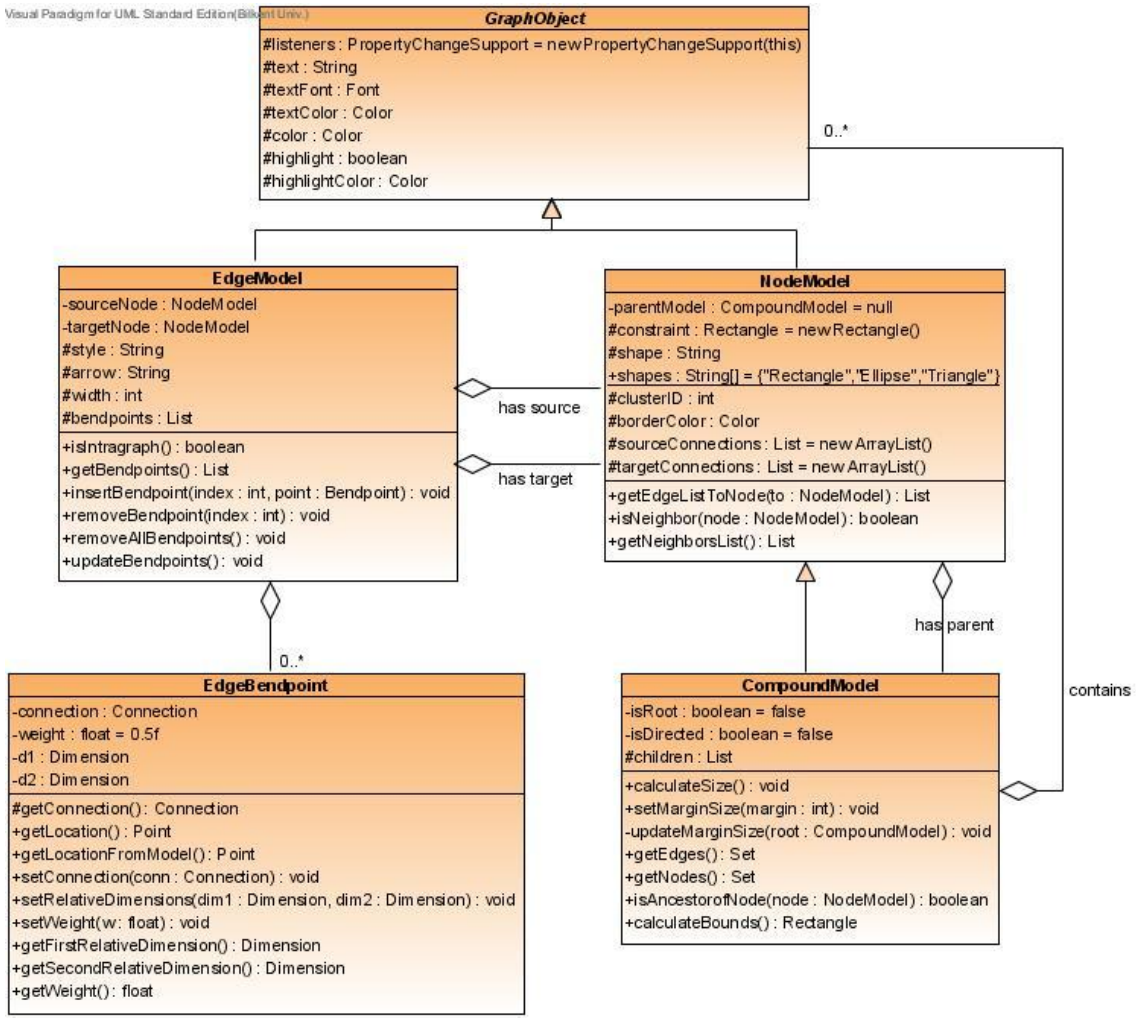

Figure 2 A UML Class Diagram illustrating the compound graph model used in Chisio Editor

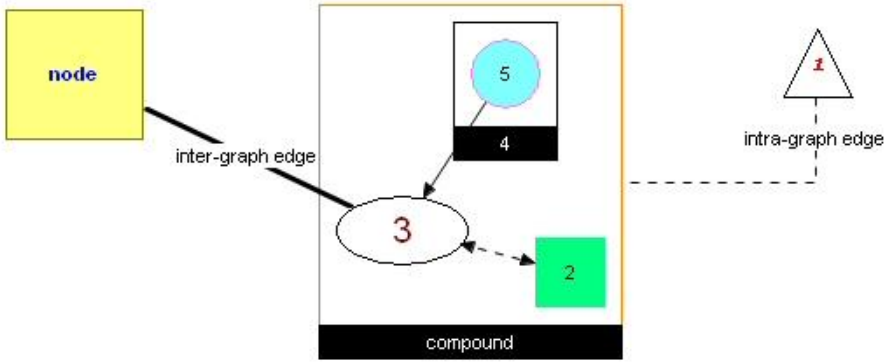

Figure 3 Basics of drawing in Chisio Editor

Chapter 2

Customizing Chisio Editor for a Specific Application

In this chapter, we describe how one can customize Chisio Editor for their specific application through programming on to the existing Chisio Editor framework.

One can customize Chisio Editor for their specific needs by adding new node/edge types or by modifying existing nodes/edges (UI and attributes). In addition, you may customize menus to add new functionality as well as modifying node and edge menus and property inspectors.

Customizing User Interface

Adding new shapes

There are several shapes available for nodes in Chisio Editor: *Rectangle*, *Circle* and *Triangle*. These shapes may be sufficient for some applications but many applications will require different types of shapes for drawing graphs of your own. In this part, we provide an example on how you can add a new shape to Chisio Editor nodes.

If you want to put an image as a figure, then you should create a figure which draws an image file. For example suppose there is an image file "node.bmp" in C: drive. We use it as a node shape in the following figure class.

```
import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;
import org.eclipse.swt.graphics.Image;

public class ImageFigure extends Figure
{
    public ImageFigure()
    {
        super();
    }

    public ImageFigure(Rectangle rect)
    {

        setBounds(rect);
    }

    protected void paintFigure(Graphics g)
    {
        Rectangle r = getParent().getBounds().getCopy();
```

```
        setBounds(r);
        Image image = new Image(null, "C:\node.bmp");
        g.drawImage(image, r.getLocation());
    }
}
```

But if you want to draw your own shape, then create your own figure by providing the code for painting it. Our example shape is a *diamond*.

First, we need to write a figure class for the new diamond shape, which extends from the existing class `Figure`. Then we need to override the `paintFigure` method as shown below:

```
public class DiamondFigure extends Figure
{
    public DiamondFigure(Rectangle rect)
    {
        setBounds(rect);
    }

    protected void paintFigure(Graphics g)
    {
        Rectangle r = getParent().getBounds().getCopy();
        setBounds(r);

        PointList points = new PointList();
        points.addPoint(new Point(r.x + r.width / 2, r.y));
        points.addPoint(new Point(r.x + r.width, r.y + r.height / 2));
        points.addPoint(new Point(r.x + r.width / 2, r.y + r.height));
        points.addPoint(new Point(r.x, r.y + r.height / 2));

        g.fillPolygon(points);
        g.drawPolygon(points);
    }
}
```

Above code segment implements a figure that draws a diamond. You can add this code into `NodeFigure` class as other figure classes for different shapes do (`RectangleFigure, EllipseFigure` and `TriangleFigure` classes).

Now we need to assign a name string to our new shape by adding it into the static array in `NodeModel` as shown below:

```
public static String[] shapes = {
    "Rectangle",
    "Ellipse",
    "Triangle",
    "Diamond"};
```

`NodeFigure` class has an `updateShape` method. This method decides which figure will be drawn. So we need to add our diamond figure into this method. In addition, we must associate our newly added name string to our new diamond figure.

```
public void updateShape(String s)
{
    this.shape = s;
    this.removeAll();
```

```
    if (shape.equals(NodeModel.shapes[0]))
    {
        add(new RectangleFigure(getBounds()));
    }
    else if (shape.equals(NodeModel.shapes[1]))
    {
        add(new EllipseFigure(getBounds()));
    }
    else if (shape.equals(NodeModel.shapes[2]))
    {
        add(new TriangleFigure(getBounds()));
    }
    else if (shape.equals(NodeModel.shapes[3]))
    {
        add(new DiamondFigure(getBounds()));
    }

    add(label);
}
```

Here, we must be careful about the index of the newly added shape name `"Diamond"`. It is in the 4th place of the static array, so if shape's type is the same with the element in the 4th place of static array (`NodeModel.shapes[3]`), this means that shape is a diamond and we create our new diamond figure.

By updating this method, we have completed addition of a new shape to Chisio Editor nodes. See **Figure 4** for examples of the new node shape.

Figure 4 Sample UIs for two diamond shaped nodes.

Adding new attributes

Each graph object (simple node, compound node or edge) in Chisio Editor has several attributes. Existing attributes may be sufficient for drawings of many types of graphs. However specific applications are most likely to have other attributes. For example, one may want to maintain a dynamic size for the arrow heads of directed edges.

We first need to update the `EdgeModel` class to handle such a new attribute for edges. Specifically we need to add a field named `arrowSize` (say, with type `int`). In addition, a default value for this new field is needed (`DEFAULT_ARROW_SIZE`). Default values are defined in the "Class Variables" section of each class.

Furthermore, a constant variable is needed for firing the property change. When we change the arrow size, this change is to be propagated to the corresponding edit-part. So, an identifier is defined in "Class Constants" section (`P_ARROW_SIZE`).

As usual, operation setter and getter methods are written for the newly created field. But in here, setter method fires the property change mentioned earlier.

```
public class EdgeModel extends GraphObject
{
    private NodeModel sourceNode;
    private NodeModel targetNode;
    protected String style;
```

```
    protected String arrow;
    protected int width;
    protected List bendpoints;
    protected int arrowSize;

    /**
     * Constructor
     */
    public EdgeModel()
    {
        super();
        this.text = DEFAULT_TEXT;
        this.textFont = DEFAULT_TEXT_FONT;
        this.textColor = DEFAULT_TEXT_COLOR;
        this.color = DEFAULT_COLOR;
        this.style = DEFAULT_STYLE;
        this.arrow = DEFAULT_ARROW;
        this.width = DEFAULT_WIDTH;
        this.bendpoints = new ArrayList();
        this.arrowSize = DEFAULT_ARROW_SIZE;
    }

    ...

    public int getArrowSize()
    {
        return arrowSize;
    }

    public void setArrowSize(int arrowSize)
    {
        this.arrowSize = arrowSize;
        firePropertyChange(P_ARROW_SIZE, null, arrow_size);
    }

    ...

// ----------------------------------------------------------------------------
// Section: Class Variables
// ----------------------------------------------------------------------------
    public static String DEFAULT_TEXT = "";

    public static Font DEFAULT_TEXT_FONT =
        new Font(null, new FontData("Arial", 8, SWT.NORMAL));

    ...

    public static int DEFAULT_WIDTH = 1;

    public static int DEFAULT_ARROW_SIZE = 1;

// ----------------------------------------------------------------------------
// Section: Class Constants
// ----------------------------------------------------------------------------
    public static final String P_STYLE = "_style";
    public static final String P_ARROW = "_arrow";
    public static final String P_WIDTH = "_width";
    public static final String P_BENDPOINT = "_bendpoint";
    public static final String P_ARROW_SIZE = "_arrowSize";
}
```

The model including the new attribute is now ready. Let's modify the figure part accordingly now. For this, we must add a new field and an update method to set this field and change the UI. We might also need to update the `paintFigure` method, which draws the UI.

```
public class EdgeFigure extends PolylineConnection
{
    ...

    boolean highlight;
    Color highlightColor;
    int arrowSize;

    /**
     * Constructor
     */
    public EdgeFigure(String text,
        Font textFont,
        Color textColor,
        Color color,
        String style,
        String arrow,
        int width,
        Color highlightColor,
        boolean highlight,
        int arrowSize)
    {
        super();

        ...

        updateWidth(width);
        updateHighlightColor(highlightColor);
        updateArrowSize(arrowSize);
    }

    ...

    public void updateArrowSize(int newArrowSize)
    {
        // Do the operations for setting the arrow size
    }

    ...
}
```

Now that we have updated the model and figure parts, we must connect them to each other. We need to update the edit-part for propagating the changes in the model to the figure (UI). The edit-part of an edge is implemented in class `ChsEdgeEditPart`. We must update `createFigure` and `propertyChange` methods in this class.

```
protected IFigure createFigure()
{
    EdgeModel model = getEdgeModel();
    EdgeFigure eFigure = new EdgeFigure(model.getText(),
        model.getTextFont(),
        model.getTextColor(),
        model.getColor(),
        model.getStyle(),
```

```
            model.getArrow(),
            model.getWidth(),
            model.getHighlightColor(),
            model.isHighlight(),
            model.getArrowSize());

        eFigure.updateHighlight(
            (HighlightLayer) getLayer(HighlightLayer.HIGHLIGHT_LAYER),
            getEdgeModel().isHighlight());

        ...
    }

    public void propertyChange(PropertyChangeEvent evt)
    {
        if (evt.getPropertyName().equals(EdgeModel.P_TEXT))
        {
            ((EdgeFigure)figure).updateText((String) evt.getNewValue());
        }

        ...

        else if (evt.getPropertyName().equals(EdgeModel.P_HIGHLIGHT))
        {
            ((EdgeFigure)figure).updateHighlight(
                (Layer) getLayer(HighlightLayer.HIGHLIGHT_LAYER));
        }
        else if (evt.getPropertyName().equals(EdgeModel.P_HIGHLIGHTCOLOR))
        {
            ((EdgeFigure)figure).
                updateHighlightColor((Color) evt.getNewValue());
        }
        else if (evt.getPropertyName().equals(EdgeModel.P_ARROW_SIZE))
        {
            ((EdgeFigure)figure).updateArrowSize((Integer) evt.getNewValue());
        }
    }
```

Thus we have completed adding a new attribute into Chisio Editor edges. In the next part, we will make the necessary changes to expose this new attribute to the users through the edge inspector window.

Modifying inspector windows

The implementation of object property inspector windows in Chisio Editor are based on a class named `Inspector` and specific inspectors extend this class (e.g. `EdgeInspector`). If you want to add a new item to the edge inspector, first you must add it into `EdgeInspector` class as follows. We will just change the constructor of the class.

```
private EdgeInspector(GraphObject model, String title, ChisioMain main)
{
    super(model, title, main);

    TableItem item = addRow(table, "Name");
    item.setText(1, model.getText());

    ...

    item = addRow(table, "Width");
    item.setText(1, "" + ((EdgeModel) model).getWidth());
```

```
    item = addRow(table, "Arrow Size");
    item.setText(1, "" + ((EdgeModel) model).getArrowSize());

    createContents(shell);

    ...
}
```

Here we add a new row with attribute name "Arrow Size", whose value is taken from the corresponding model object. Now, we need to update `createContents` method of `Inspector` class to add interactivity to our newly added attribute. We already had five types of objects in our edge inspector tables: *Combo*, *Text*, *Number*, *Color* and *Font*. Our new attribute's value is a number. So we need to add it into the if-statement, which is tagged with "NUMBER".

```
public void createContents(final Shell shell)
{
    // Create an editor object to use for text editing
    final TableEditor editor = new TableEditor(table);
    editor.horizontalAlignment = SWT.LEFT;
    editor.grabHorizontal = true;

    // Use a selection listener to get seleceted row
    table.addSelectionListener(new SelectionAdapter()
    {
        public void widgetSelected(SelectionEvent event)
        {
            // Dispose any existing editor
            Control old = editor.getEditor();

            ...

            if (item != null)
            {
                // COMBO
                if (item.getText().equals("Style")
                    || item.getText().equals("Arrow")
                    || item.getText().equals("Shape"))
                {
                    ...
                }
                // TEXT
                else if (item.getText().equals("Name"))
                {
                    ...
                }
                // NUMBER
                else if (item.getText().equals("Margin")
                    || item.getText().equals("Cluster ID")
                    || item.getText().equals("Width")
                    || item.getText().equals("Arrow Size"))
                {
                    ...
                }
                // COLOR
                else if (item.getText().equals("Border Color")
                    || item.getText().equals("Color")
                    || item.getText().equals("Highlight Color"))
                {
                    ...
```

```
                }
                // FONT
                else if (item.getText().equals("Text Font"))
                {
                    ...
                }

                table.setSelection(-1);
            }
        }
    });

    ...
}
```

We have added new attribute into our inspector window (**Figure 5**). When "OK" button is pressed, values are transferred to the model. So we need to update the listener for the "OK" button to add the newly added arrow size attribute as below.

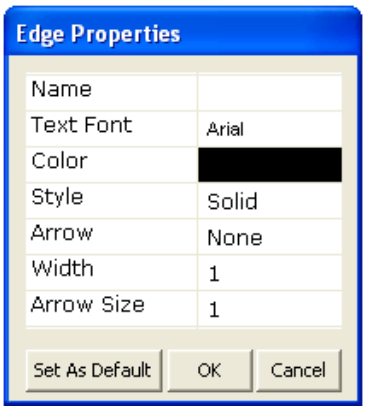

Figure 5 Sample screenshot of the Edge Inspector after addition of the new attribute "Arrow Size".

```
okButton.addSelectionListener(new SelectionAdapter()
{
    public void widgetSelected(final SelectionEvent e)
    {
        TableItem[] items = table.getItems();

        for (TableItem item : items)
        {
            if (item.getText().equals("Name"))
            {
                model.setText(item.getText(1));
            }

            ...

            else if (item.getText().equals("Margin"))
            {
                new ChangeMarginAction((CompoundModel)model,
                    Integer.parseInt(item.getText(1))).run();
            }
            else if (item.getText().equals("Arrow Size"))
            {
                ((EdgeModel) model).
                    setArrowSize(Integer.parseInt(item.getText(1)));
            }
        }
```

```
        shell.close();
    }
});
```

Furthermore, we need to update `setAsDefault` method in `EdgeInspector` class to change the default arrow size with this new value.

```
public void setAsDefault()
{
    TableItem[] items = table.getItems();

    for (TableItem item : items)
    {
        if (item.getText().equals("Name"))
        {
            EdgeModel.DEFAULT_TEXT = item.getText(1);
        }

        ...

        else if (item.getText().equals("Width"))
        {
            EdgeModel.DEFAULT_WIDTH = Integer.parseInt(item.getText(1));
        }
        else if (item.getText().equals("Arrow Size"))
        {
            EdgeModel.DEFAULT_ARROW_SIZE = Integer.parseInt(item.getText(1));
        }
    }
}
```

Viola! We have now completed adding a new item into an inspector.

Changing defaults for nodes and edges

There are default values for the creation of simple and compound nodes and edges. These values can be changed easily by updating a few lines.

Defaults for nodes are kept in the `NodeModel` class in the "Class Variables" section. By changing these variables, we can change the defaults for nodes.

```
// ----------------------------------------------------------------------------
// Section: Class Variables
// ----------------------------------------------------------------------------
    public static Dimension DEFAULT_NODE_SIZE = new Dimension(40, 40);

    public static String DEFAULT_TEXT = "Node";

    public static Font DEFAULT_TEXT_FONT =
        new Font(null, new FontData("Arial", 8, SWT.NORMAL));

    public static Color DEFAULT_TEXT_COLOR = ColorConstants.black;

    public static Color DEFAULT_COLOR = new Color(null, 14, 112, 130);

    public static Color DEFAULT_BORDER_COLOR = new Color(null, 14, 112, 130);

    public static String DEFAULT_SHAPE = shapes[0];
```

Compound node and edge default values may be changed similarly under the "Class Variables" section of `CompoundModel` and `EdgeModel` classes, respectively.

File Operations

Persistence of newly added attributes requires special attention. Below you will find what you need to do so these attributes can be saved and loaded back properly on disk.

Integrating new attributes into save operation

We have added the "arrow size" attribute for edges earlier on. Now, we will add support for this attribute in the save operation.

There is a `GraphMLWriter` class for saving our graphs as a ".graphml" file. We need to update this class.

First, we must define a `KeyType` for arrow size attribute (`arrowSizeKey`). Also we need to add code into `writeXMLFile` and `createEdge` methods as shown below. If we had added a new parameter for nodes, then we should have changed `createNode` method as well.

```
public class GraphMLWriter extends XMLWriter
{
    HashMap hashMap = new HashMap();

    GraphmlType newGraphml;

    KeyType xKey;

    ...

    KeyType highlightColorKey;

    KeyType arrowSizeKey;

    public Object writeXMLFile(CompoundModel root)
    {
        // create a new graphml file
        GraphmlDocument newGraphmlDoc = GraphmlDocument.Factory.newInstance();
        newGraphml = newGraphmlDoc.addNewGraphml();

        // define the keys that will be used in xml file
        xKey = newGraphml.addNewKey();
        xKey.setId("x");
        xKey.setFor(KeyForType.NODE);
        xKey.setAttrName("x");
        xKey.setAttrType(KeyTypeType.INT);

        ...

        highlightColorKey = newGraphml.addNewKey();
        highlightColorKey.setId("highlightColor");
        highlightColorKey.setFor(KeyForType.ALL);
        highlightColorKey.setAttrName("highlightColor");
        highlightColorKey.setAttrType(KeyTypeType.STRING);

        arrowSizeKey = newGraphml.addNewKey();
        arrowSizeKey.setId("arrowSize");
        arrowSizeKey.setFor(KeyForType.NODE);
        arrowSizeKey.setAttrName("arrowSize");
```

```
        arrowSizeKey.setAttrType(KeyTypeType.INT);

        // create a new graph with our root node recursively
        GraphType newGraph = newGraphml.addNewGraph();
        createTree(newGraph, root, "");

        return newGraphmlDoc;
    }

    public void createEdge(EdgeType newEdge, EdgeModel model)
    {
        // write edge's properties into file
        DataType colorData = newEdge.addNewData();
        colorData.setKey(colorKey.getId());
        RGB rgb = model.getColor().getRGB();
        colorData.set(XmlString.Factory.newValue(rgb.red + " " +
            rgb.green + " " + rgb.blue));

        ...

        if (model.isHighlight())
        {
            DataType highlightColorData = newEdge.addNewData();
            highlightColorData.setKey(highlightColorKey.getId());
            rgb = model.getHighlightColor().getRGB();
            highlightColorData.set(XmlString.Factory.
                newValue(rgb.red + " " + rgb.green + " " + rgb.blue));
        }

        DataType arrowSizeData = newEdge.addNewData();
        arrowSizeData.setKey(arrowSizeKey.getId());
        arrowSizeData.set(XmlString.Factory.
            newValue("" + model.getArrowSize()));
    }
}
```

That's all. We now have support for our new attribute on save.

Integrating new attributes into load operation

Now that we know how to save graphs with newly introduced attribute "arrow size", let's now integrate this attribute into the load operation. For this purpose, we need to update `readEdge` method in `GraphMLReader` class as shown below. If we had added a new parameter for nodes, then we should have updated the `readNode` method as well.

```
public void readEdge(EdgeType edge, EdgeModel model)
{
    int width = -1;
    String style = null;

    ...

    int arrowSize = -1;

    // read edge's properties
    DataType[] datas = edge.getDataArray();

    for (int i = 0; i < datas.length; i++)
    {
        DataType data = datas[i];
```

```
        if (data.getKey().equalsIgnoreCase("color"))
        {
            color = data.newCursor().getTextValue();
        }

        ...

        else if (data.getKey().equalsIgnoreCase("highlightColor"))
        {
            highlightColor = data.newCursor().getTextValue();
        }
        else if (data.getKey().equalsIgnoreCase("arrowSize"))
        {
            arrowSize = Integer.parseInt(data.newCursor().getTextValue());
        }
    }

    ...

    if (highlightColor != null)
    {
        String[] rgb = highlightColor.split(" ");
        int r = Integer.parseInt(rgb[0]);
        int g = Integer.parseInt(rgb[1]);
        int b = Integer.parseInt(rgb[2]);
        model.setHighlightColor(new Color(null, r, g, b));
        model.setHighlight(true);
    }

    if (arrowSize > 0)
    {
        model.setArrowSize(arrowSize);
    }
}
```

By reading the `arrowSize` field in ".graphml" files, we have added the support for loading newly created attribute "arrow size".

Reading other file formats

Chisio Editor uses `GraphML` file format for persistent storage of graphs. There are two classes `GraphMLReader` and `GraphMLWriter` for load and save operations, which are extended from abstract classes `XMLReader` and `XMLWriter`.

If you would like to support other file formats such as GXL, GML or your own XML-based format, you must extend `XMLReader` and `XMLWriter` classes accordingly. Suppose we want to support ".gxl" files instead.

```
public class GXLReader extends XMLReader
{
    public CompoundModel readXMLFile(File xmlFile)
    {
        // parse GXL file in here and return the read root graph
    }
}

public class GXLWriter extends XMLWriter
{
    public Object writeXMLFile(CompoundModel root)
```

```
    {
        // create GXL file from root graph return the file content
    }
}
```

After writing new classes named `GXLReader` and `GXLWriter` similar to their ".graphml" counterpart, we need to change `LoadAction` and `SaveAction` classes as below.

```
public class LoadAction extends Action
{
    ...

    public void run()
    {
        ...

        File xmlfile = new File(filename);

        XMLReader reader = new GXLReader();
        CompoundModel root = reader.readXMLFile(xmlfile);

        ...
    }
}

public class SaveAction extends Action
{
    ...

    public void run()
    {
        ...

        BufferedWriter xmlFile =
            new BufferedWriter(new FileWriter(fileName));

        XMLWriter writer = new GXLWriter();
        xmlFile.write(writer.writeXMLFile(root).toString());
        xmlFile.close();

        ...
    }
}
```

In the `run` method of `LoadAction` class, we simply change `GraphMLReader` to `GXLReader` and in the run method of `SaveAction` class, change `GraphMLWriter` to `GXLWriter`. That's all.

Menu Operations

Each operation in the toolbar menu, pop-up menus or the main menu has an associated `Action`. So, if you would like to add a new menu item, you must first create an associated action for it.

For example, let's suppose we want to add a new operation for finding neighbors of a selected node and highlight them. We can write a new action for this operation as follows. Icon for this action is set in the constructor method.

```
import org.eclipse.jface.action.Action;
```

```
public class HighlightNeighborsAction extends Action
{
    ChisioMain main;

    /**
     * Constructor
     */
    public HighlightNeighborsAction(ChisioMain main)
    {
        super("Highlight Neighbors");
        this.setToolTipText("Highlight Neighbors");
        setImageDescriptor(ImageDescriptor.createFromFile(
            ChisioMain.class,
            "icon/highlight-neighbors.gif"));
        this.main = main;
    }

    public void run()
    {
        ScrollingGraphicalViewer viewer = main.getViewer();
        // Iterates selected objects; for each selected objects, highlights them
        Iterator selectedObjects =
            ((IStructuredSelection) viewer.getSelection()).iterator();

        if (selectedObjects.hasNext())
        {
            Object model = ((EditPart)selectedObjects.next()).getModel();

            if (model instanceof NodeModel)
            {
                Iterator<NodeModel> neighbors =
                    ((NodeModel) model).getNeighborsList().iterator();

                while (neighbors.hasNext())
                {
                    NodeModel next = neighbors.next();
                    next.setHighlightColor(main.higlightColor);
                    next.setHighlight(true);
                }
            }
        }
    }
}
```

Now, we can use this action to create a new menu item.

New main menu item

We must update `createBarMenu` method in `TopMenuBar` class for adding this action to the main menu. Suppose the new operation is to be added under the "View" menu.

```
public static MenuManager createBarMenu(ChisioMain main)
{
    chisio = main;

    MenuManager menuBar = new MenuManager("");
    MenuManager fileMenu = new MenuManager("&File");
    MenuManager editMenu = new MenuManager("&Edit");
    MenuManager viewMenu = new MenuManager("&View");
    ...
```

```
    viewMenu.add(new Separator());
    viewMenu.add(new HighlightSelectedAction(chisio));
    viewMenu.add(new HighlightNeighborsAction(chisio));
    viewMenu.add(new RemoveHighlightFromSelectedAction(chisio));
    viewMenu.add(new RemoveHighlightsAction(chisio));

    ...
}
```

New pop menu item

We must update `createActions` method in `PopupManager` class for adding this new operation to popup menu. This is an action for nodes/compound nodes. So we must add it to `NODE-COMPOUND POPUP` part of the code.

```
public void createActions(IMenuManager manager)
{
    EditPart ep = main.getViewer().findObjectAt(clickLocation);

    if (ep instanceof RootEditPart)
    {
        // GRAPH POPUP
        manager.add(new ZoomAction(main, 1, clickLocation));
        manager.add(new ZoomAction(main, -1, clickLocation));
        manager.add(new RemoveHighlightsAction(main));
        manager.add(new LayoutInspectorAction(main));
        manager.add(new InspectorAction(main, true));
        main.getViewer().select(ep);
    }
    else if (ep instanceof NodeEditPart)
    {
        // NODE-COMPOUND POPUP
        manager.add(new CreateCompoundFromSelectedAction(main));
        manager.add(new HighlightSelectedAction(main));
        manager.add(new HighlightNeighborsAction(main));
        manager.add(new RemoveHighlightFromSelectedAction(main));
        manager.add(new DeleteAction(main.getViewer()));
        manager.add(new InspectorAction(main, false));
    }
    else if (ep instanceof ChsEdgeEditPart)
    {
        // EDGE POPUP
        manager.add(new HighlightSelectedAction(main));
        manager.add(new RemoveHighlightFromSelectedAction(main));
        manager.add(new DeleteAction(main.getViewer()));
        manager.add(new InspectorAction(main, false));
    }
}
```

New toolbar item

We must update `ToolbarManager` class for adding this operation to the toolbar menu. The icon for this item is already set in the constructor of `HighlightNeighborsAction` class.

```
public ToolbarManager(int style, ChisioMain main)
{
    super(style);

    add(new NewAction("New", main));
    add(new LoadAction(main));
```

```
    add(new SaveAction(main));

    ...

    add(new HighlightNeighborsAction(main));

    ...

    add(new SugiyamaLayoutAction(main));
    add(new Separator());

    update(true);
}
```

Chapter 3

Adding New Layout Operations into Chisio Editor

In this chapter, we describe how one can integrate a new layout operation into the existing layout architecture of Chisio Editor.

Chisio currently supports several layout styles from the basic spring embedder to hierarchical (Sugiyama) layout to compound spring embedder to circular layout via Chisio Layout package. For details on these layout operations, please refer to Chisio Layout Programmer's Guide.

Chisio layout architecture

For Chisio layout architecture please refer to Chisio Layout Programmer's Guide/Overview/System Architecture.

Each layout operation runs on a separate thread for more robust animation as well as for allowing cancellation. Also, all layout operations can be run without any need for Chisio Editor window (i.e. graphical user interface). This facilitates running layout in batch mode, especially useful for testing purposes.

Adding new layout algorithms

For existing layout styles, adding new layout styles and customizing the existing ones please refer to the Chisio Layout Programmer's Guide/Layout & Customization.

After updating layout options pack which gives us the ability to parameterize our layouts. We can easily change the layout properties window by adding a new tab for our new style as well. Our new layout style will use the values in layout options pack, which will be set through a new tab in the layout properties window as described below.

```
public class LayoutInspector extends Dialog
{
    ...

    protected void createContents()
    {
        ...

        // Create the UI for new layout. Buttons, sliders, checkboxes etc.
```

```
        ...
    }

    public void storeValuesToOptionsPack()
    {
        LayoutOptionsPack lop = LayoutOptionsPack.getInstance();

        // General
        lop.getGeneral().setAnimationPeriod(animationPeriod.getSelection());
        lop.getGeneral().setAnimationOnLayout(
            animateOnLayoutButton.getSelection());

        ...

        // Random
        lop.getRandom().set..
        lop.getRandom().set..
    }

    public void setInitialValues()
    {
        LayoutOptionsPack lop = LayoutOptionsPack.getInstance();

        // General
        animationPeriod.setSelection(lop.getGeneral().getAnimationPeriod());
        animateDuringLayoutButton.setSelection(
            lop.getGeneral().isAnimationDuringLayout());

        ...

        // Random

        // Take values from UI and set into layout options pack (lop)
    }

    public void setDefaultLayoutProperties(int select)
    {
        if (select == 0)
        {
            // General
            animateDuringLayoutButton.setSelection(
                AbstractLayout.DEFAULT_ANIMATION_DURING_LAYOUT);
            animationPeriod.setSelection(50);

            ...
        }

        ...

        else if (select == 7)
        {
            // Random

            // Fill the UI with the default values of layout
        }
    }
}
```

Thus, we have added a new tab for our new layout algorithm to set its associated exposed parameters.

Changing defaults for layouts

The default values for both exposed and non-exposed parameters of a layout style may be modified. For exposed parameters these refer to the defaults in Layout Properties window. For non-exposed ones, the only way to change a layout parameter is by updating the default values in the associated section of the corresponding layout class. (Please refer to the Chisio Layout Programmer's Guide/Layout & Customization/Existing Layout Styles)

Performance analysis

The performance of a particular layout algorithm could be measured by looking at certain properties (e.g. number of crossings, total area, and execution time) of a laid out drawing. For this purpose, you may use the class `org.gvt.layout.PerformanceAnalyser`.

The class `org.gvt.util.RandomGraphCreator` may be used to create random graphs for performance analysis. You may refer to the User's Guide for details on random graph creation. In addition, a good number of connected bi-planar subset of Rome graphs (http://www.dia.uniroma3.it/~gdt/gdt4/test_suite.php) have been imported and are available for batch or interactive use in GraphML format.

Chapter 4

Third-Party Software License Agreements

In this chapter, we list any third-party software license agreements.

Chisio makes use of Eclipse Graph Editing Framework (GEF) version 3.1 distributed under license Eclipse Public License – v 1.0, for drawing and editing pathways. Silk icons, which are distributed under Creative Commons Attribution 2.5 License, have been used to design icons for the toolbar.

Component	Version	By	License
Eclipse Graph Editing Framework (GEF)	3.1	Eclipse	Eclipse Public License - v 1.0
Silk Icons	-	Mark James	Creative Commons Attribution License, v 2.5
XMLBeans	-	**The Apache Software Foundation**	Apache License

References

[1] https://sourceforge.net/projects/chilay/