```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.awt.datatransfer.*;
import java.lang.reflect.*;
import java.net.*;

import gjt.*;

public class DrawGui extends Frame
{
    public DrawGui(Controller con)
    {
        super("Harpoon Prototype");
        setSize(500,400);
        myDim = this.getSize();
        myController = con;
        myCommands = new Commands();

        setLayout(new BorderLayout());

        myDrawPanel = new DrawPanel();
        this.add(BorderLayout.CENTER,myDrawPanel);

        Panel p = new Panel();
        p.add(myMessages = new TextField(40));
        this.add(BorderLayout.SOUTH,p);
        this.add(BorderLayout.NORTH,makeToolbar());

        makeMenus();
        makeTools();

        // don't pack in BorderLayout, it changes dimensions of components

        setVisible(true);

        myFigures = new Vector();
    }

    private void makeMenus()
    {
        // make menubar with three menus: file, edit, windows

        MenuBar bar = new MenuBar();

        // make file menu

        Menu fileMenu     = new Menu("File");

        MenuItem openMI   = new MenuItem(Consts.OPEN);
        MenuItem saveMI   = new MenuItem(Consts.SAVE);
        MenuItem saveasMI = new MenuItem("Save as");
        MenuItem printMI  = new MenuItem(Consts.PRINT);
        MenuItem quitMI   = new MenuItem(Consts.QUIT);

        openMI.addActionListener( myCommands.get(Consts.OPEN));
        saveMI.addActionListener( myCommands.get(Consts.SAVE));
        printMI.addActionListener(myCommands.get(Consts.PRINT));

        quitMI.addActionListener(Quitter.getInstance());

        fileMenu.add(openMI);
        fileMenu.add(saveMI);
        fileMenu.add(saveasMI);
        fileMenu.add(printMI);
        fileMenu.addSeparator();
        fileMenu.add(quitMI);

        // make figure menu
```



```
        Menu toolMenu    = new Menu("Tools");
        MenuItem rectMI    = new MenuItem(Consts.RECTANGLE);
        MenuItem ellipseMI = new MenuItem(Consts.ELLIPSE);
        MenuItem imageMI   = new MenuItem(Consts.IMAGE);
        MenuItem moveMI    = new MenuItem(Consts.MOVE);

        final NullTool nt = new NullTool();

        ellipseMI.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    myController.setTool(nt);
                }
            });


        rectMI.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    Tool t = (Tool) myTools.get(Consts.RECTANGLE);
                    myController.setTool(t);
                }
            });

        moveMI.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    Tool t = (Tool) myTools.get(Consts.MOVE);
                    myController.setTool(t);
                }
            });


        imageMI.addActionListener(new ImageLoadCommand(this,myController));

        toolMenu.add(ellipseMI);
        toolMenu.add(rectMI);
        toolMenu.add(imageMI);
        toolMenu.addSeparator();
        toolMenu.add(moveMI);

        // add all menus to menubar

        bar.add(fileMenu);
        bar.add(makeEditMenu());
        bar.add(toolMenu);
        bar.add(makeColorMenu());

        this.setMenuBar(bar);

    }


    private Menu makeColorMenu()
    {
        Menu colorMenu = new Menu("Color");
        Class c = null;
        try{
            c = Class.forName("java.awt.Color");
        }
        catch (ClassNotFoundException e)
        {
```

```
            showMessage("could not load colors "+e);
        }

        Field[] colorFields = c.getFields();

        for(int k = 0; k < colorFields.length; k++)
        {
            final Field  f    = colorFields[k];
            final String name = f.getName();
            Color tempColor = Color.red;          // some default value
            try{
                tempColor = (Color) f.get(null);  // value of field
            }
            catch(Exception e){
                continue;
            }
            final Color theColor = tempColor;

            MenuItem mi = new MenuItem(name);
            colorMenu.add(mi);
            mi.addActionListener(
                new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        showMessage("color: "+name);
                        myColor = theColor;
                    }
                });
        }
        return colorMenu;
    }

    /**
      * make the edit menu
      */

    private Menu makeEditMenu()
    {
        Menu edit         = new Menu("Edit");
        MenuItem copyMI   = new MenuItem("copy");
        MenuItem cutMI    = new MenuItem("cut");
        myPaste           = new MenuItem("paste");
        MenuItem deleteMI = new MenuItem("delete");
        MenuItem deleteAllMI = new MenuItem("delete all");

        edit.add(copyMI);
        edit.add(cutMI);
        edit.add(myPaste);
        edit.addSeparator();
        edit.add(deleteMI);
        edit.add(deleteAllMI);

        copyMI.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    doCopy();
                }
            });

        cutMI.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    doCut();
                }
```



```
            });

        deleteMI.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    delete();
                }
            });

        deleteAllMI.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    deleteAll();
                }
            });


        myPaste.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    doPaste();
                }
            });

        return edit;
    }

    /**
      * make tools used in drawing
      */

    private void makeTools()
    {
        myTools = new Hashtable();
        myTools.put(Consts.RECTANGLE, new
                    RectangleTool(myController));  // draw rectangles
        myTools.put(Consts.MOVE,new
                    MoveTool(myController));       // move figures
    }

    /**
      * make a toolbar, semi-functional
      */

    private Toolbar makeToolbar()
    {
        Toolbar toolb = new Toolbar(10,0);
        ResourceBundle bundle = ResourceBundle.getBundle("Toolbar");

        String[] toolnames = {
            "new", Consts.OPEN, Consts.SAVE, Consts.CUT,
            Consts.COPY, Consts.PASTE
        };

        for(int k=0; k < toolnames.length; k++)
        {
            String name = toolnames[k];
            URL source = this.getClass().getResource(name+".gif");
            showMessage("loading "+source);
            Image im = ImageLoader.loadImage(this,source);

            ImageButton imb = toolb.add(im);
```

```
            ActionListener al = myCommands.get(name);
            if (al != null)
            {
                imb.addActionListener(al);
            }

            // bubbles are trouble, noclassdef and null pointer problems

//          imb.setBubbleHelp(bundle.getString(name));
//          imb.setBubbleInterval(5);

        }
        return toolb;
    }

    /**
      * cut the selected figure to the clipboard
      */
    private void doCut()
    {
        Figure f = getSelectedFigure();
        doCopy();
        delete(f);
    }

    /**
      * delete a figure
      * @param f is the figure being deleted
      */
    public void delete(Figure f)
    {
        myFigures.removeElement(f);
        myDrawPanel.remove(f);
        myDrawPanel.repaint();
    }

    /**
      * delete all figures in the drawing
      * @see delete
      */

    public void deleteAll()
    {
        Enumeration e = myFigures.elements();
        while (e.hasMoreElements())
        {
            Figure f = (Figure) e.nextElement();
            myDrawPanel.remove(f);
        }
        myFigures.removeAllElements();
        myDrawPanel.repaint();
    }

    /**
      * delete the selected figure
      */

    public void delete()
    {
        delete(getSelectedFigure());
    }

    /**
      * @return the figure currently selected
      * (currently returns first figure)
      */

    public Figure getSelectedFigure()
    {
```



```
        if (myFigures.size() > 0)
        {
            return (Figure) myFigures.elementAt(0);
        }
        else
        {
            return null;
        }
    }

    /**
      * return figure containg p (first found in z-order)
      * @param p is the point searched for among all figures
      * @return the first (z-order) figure containg p, null if none found
      */

    public Figure getFigure(Point p)
    {
        Enumeration e = myFigures.elements();
        while (e.hasMoreElements())
        {
            Figure f = (Figure) e.nextElement();
            if ( f.contains(p) ) return f;
        }
        return null;
    }

    private void doCopy()
    {
        Figure f = getSelectedFigure();
        if (f != null)
        {
            FigureSelection fs = new FigureSelection(f);
            this.getToolkit().getSystemClipboard().setContents(fs,fs);
        }
    }

    private void doPaste()
    {
        Transferable content =
            this.getToolkit().getSystemClipboard().getContents(this);
        if (content != null)
        {
            try{
                Figure f = (Figure)
                    content.getTransferData(FigureSelection.ComponentFlavor);
                addFigure(f,true);
            }
            catch (Exception e){
                showMessage("paste fail "+
                  FigureSelection.ComponentFlavor.getHumanPresentableName());
            }
        }
    }

    /**
      * add a figure to the drawing, can batch adds using false for redraw
      * @param f is the figure added
      * @param redraw is true if repainting occurs, false if no repaing
      */

    public void addFigure(Figure f, boolean redraw)
    {
        myFigures.addElement(f);
        myDrawPanel.add(f);
        if (redraw)
        {
            myDrawPanel.repaint();
        }
```

```
    }

    /**
      * used for diagnostics, errors, etc. for user
      *
      * @param message displayed to the user
      */

    public void showMessage(String message)
    {
        myMessages.setText(message);
    }

    /**
      * @return a vector of figures in the drawing (shared with drawing)
      */

    public Vector getFigures()
    {
        return myFigures;
    }

    /**
      * add figures to drawing, existing figures are not removed
      *
      * @param figs is a vector of figures added to drawing
      */

    public void addFigures(Vector figs)
    {
        Enumeration e = figs.elements();

        while (e.hasMoreElements())
        {
            addFigure((Figure)e.nextElement(),false);
        }
        myDrawPanel.repaint();
    }

    public Container getDrawable()
    {
        return myDrawPanel;
    }


    public void update(Graphics g)
    {
        paint(g);
    }

    public Color getColor()
    {
        return myColor;
    }

    private DrawPanel  myDrawPanel;
    // private DoubleBufferedContainer  myDrawPanel;
    private Color      myColor = Color.white;
    private Dimension  myDim;
    private Vector     myFigures;
    private Hashtable  myTools;
    private Controller myController;
    private TextField  myMessages;
    private MenuItem   myPaste;
    private Commands   myCommands;

    public static void main(String args[])
    {
        Controller con = new Controller();
```



```
        DrawGui gui = new DrawGui(con);
        con.addGui(gui);
    }

    class Commands
    {
        Commands()
        {
            myCommands = new Hashtable();

            myCommands.put(Consts.OPEN,
                           new LoadCommand(DrawGui.this,myController));

            myCommands.put(Consts.SAVE,
                           new SaveCommand(DrawGui.this,myController));

            myCommands.put(Consts.PRINT,
                           new PrintCommand(DrawGui.this,myController));

        }

        ActionListener get(String s)
        {
            return (ActionListener) myCommands.get(s);
        }

        Hashtable myCommands;
    }


}
```

```
import java.awt.datatransfer.*;
import java.io.*;

public class FigureSelection implements Transferable, ClipboardOwner
{
    FigureSelection(Figure f)
    {
        myFigure = (Figure) f.clone();
    }

    public synchronized DataFlavor[] getTransferDataFlavors()
    {
        return myFlavors;
    }

    public synchronized Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException, IOException
    {
        if (flavor.equals(ComponentFlavor))
        {
            return myFigure;
        }
        else
        {
            throw new UnsupportedFlavorException(flavor);
        }
    }

    public boolean isDataFlavorSupported(DataFlavor flavor)
    {
        return flavor.equals(ComponentFlavor);
    }

    public void lostOwnership(Clipboard c, Transferable t)
    {
        // nothing to do
    }

    private DataFlavor[] myFlavors = {ComponentFlavor};
    private Figure myFigure;

    static public DataFlavor ComponentFlavor;
    static
    {
        try{
            ComponentFlavor = new DataFlavor(
                Class.forName("java.awt.Component"), "AWT Component");
        }
        catch(ClassNotFoundException e){
            System.err.println("error in constructing ComponentFlavor");
            e.printStackTrace();
        }
    }
}
```

```
import java.awt.FileDialog;
import java.awt.event.*;
import java.awt.*;

/**
 * open a file dialog to read the name of an image
 *
 * @author Owen Astrachan
 */

public class LoadCommand implements ActionListener
{
    public LoadCommand(Frame f, Controller c)
    {
        myControl = c;
        myFrame   = f;
        myDialog = new
            FileDialog(myFrame, "Harpoon Load", FileDialog.LOAD);
    }

    public void actionPerformed(ActionEvent ev)
    {
        String filename = getFilename();
        if (filename == null) return;

        myControl.load(filename);
    }

    private String getFilename()
    {

        myDialog.setVisible(true);
        String retval = null;

        if (myDialog.getFile() != null)
        {
            retval = myDialog.getDirectory() + myDialog.getFile();
        }
        myDialog.setVisible(false);
        return retval;
    }

    private Controller myControl;
    private Frame      myFrame;
    private FileDialog myDialog;
}
```

```
import java.awt.event.*;
import java.awt.*;

import gjt.*;

/**
  * this class moves a figure, the ideas are from <em>Graphic Java,
  * Mastering the AWT</em>, by David Geary, second edition, in
  * particular see the DoubleBufferedContainer example and the class
  * LightweightDragger on page 497
  *
  * @author Owen Astrachan
  */

public class MoveTool extends ToolAdapter
{
    public MoveTool(Controller c)
    {
        super(c);
    }

    public void setActive(boolean status)
    {
        if (status == true)
        {
            myController.getDrawingArea().addMouseListener(this);
            myController.getDrawingArea().addMouseMotionListener(this);
        }
        else
        {
            myController.getDrawingArea().removeMouseListener(this);
            myController.getDrawingArea().removeMouseMotionListener(this);
        }
    }

    public void mousePressed(MouseEvent e)
    {
        myLast = new Point(e.getX(), e.getY());
        myFigure = myController.getFigure(myLast);
        myIsDragging = true;
    }

    public void mouseReleased(MouseEvent e)
    {
        myIsDragging = false;
    }

    public void mouseClicked(MouseEvent e)
    {
        myIsDragging = false;
    }

    public void mouseDragged(MouseEvent e)
    {
        if (myIsDragging && myFigure != null)
        {
            Point loc = myFigure.getLocation();
            Point nextLoc = new Point(
                loc.x + e.getX() - myLast.x,
                loc.y + e.getY() - myLast.y);

            Rectangle bounds = myFigure.getBounds();
            myFigure.setLocation(nextLoc);
            bounds.add(myFigure.getBounds());

            // why is this here, needed in 1.1.5, not in 1.1.3 ?
            // this is repainting based on a clip of damaged region

            myFigure.getParent().repaint(bounds.x,bounds.y,
```



```
                                         bounds.width,bounds.height);

            myLast.x = e.getX();
            myLast.y = e.getY();
        }
    }

    private boolean myIsDragging = false;
    private Point myLast;
    private Figure myFigure;
}
```

```
public class NullTool extends ToolAdapter
{
    public NullTool()
    {

    }

    public void setActive(boolean status)
    {

    }
}
```

```
import java.awt.event.*;

public abstract class ToolAdapter implements
                      Tool, MouseListener, MouseMotionListener
{

    protected ToolAdapter()
    {
    }
    protected ToolAdapter(Controller c)
    {
        myController = c;
    }

   // MouseListener events

    public void mousePressed (MouseEvent evt) {}
    public void mouseReleased(MouseEvent evt) {}
    public void mouseEntered (MouseEvent evt) {}
    public void mouseExited  (MouseEvent evt) {}
    public void mouseClicked (MouseEvent evt) {}

    // MouseMotionListener events

    public void mouseDragged(MouseEvent evt)  {}
    public void mouseMoved  (MouseEvent evt)  {}

    public abstract void setActive(boolean status);

    // ---------

    protected Controller myController;
}
```

```
import java.awt.*;
import java.awt.datatransfer.*;
import java.io.*;

public abstract class Figure
    extends Component implements Serializable, Cloneable
{
    public abstract Object clone();
}
```