```
# Executables
OSTYPE = $(shell uname -s)
JAVAC = javac
JAVA = java
A2PS = a2ps-utf8
GHOSTVIEW = gv
DOCP = javadoc
ARCH = zip
PS2PDF = ps2pdf -sPAPERSIZE=a4
DATE = $(shell date +%Y-%m-%d)
# Options de compilation
#CFLAGS = -verbose
CFLAGS =
CLASSPATH=.

JAVAOPTIONS = --verbose

PROJECT=TP_FiguresEditor
# nom du fichier d'impression
OUTPUT = $(PROJECT)
# nom du répertoire ou se situera la documentation
DOC = doc
# lien vers la doc en ligne du JDK
WEBLINK = "http://docs.oracle.com/javase/6/docs/api/"
# lien vers la doc locale du JDK
LOCALLINK = "file:///Users/davidroussel/Documents/docs/java/api/"
# nom de l'archive
ARCHIVE = $(PROJECT)
# format de l'archive pour la sauvegarde
ARCHFMT = zip
# Répertoire source
SRC = src
# Répertoire bin
BIN = bin
# Répertoire Listings
LISTDIR = listings
# Répertoire Archives
ARCHDIR = archives
# Répertoire Figures
FIGDIR = graphics
# noms des fichiers sources
MAIN = Editor ShapesDemo2D
SOURCES = $(foreach name, $(MAIN), $(SRC)/$(name).java) \
$(SRC)/figures/Figure.java \
$(SRC)/figures/Drawing.java \
$(SRC)/figures/Circle.java \
$(SRC)/figures/Ellipse.java \
$(SRC)/figures/Rectangle.java \
$(SRC)/figures/RoundedRectangle.java \
$(SRC)/figures/Polygon.java \
$(SRC)/figures/creationListeners/AbstractCreationListener.java \
$(SRC)/figures/creationListeners/RectangularShapeCreationListener.java \
$(SRC)/figures/creationListeners/RoundedRectangleCreationListener.java \
$(SRC)/figures/creationListeners/PolygonCreationListener.java \
$(SRC)/figures/creationListeners/package-info.java \
$(SRC)/figures/enums/FigureType.java \
$(SRC)/figures/enums/LineType.java \
$(SRC)/figures/enums/PaintToType.java \
$(SRC)/figures/enums/package-info.java \
$(SRC)/figures/package-info.java \
$(SRC)/filters/FigureFilter.java \
$(SRC)/filters/FigureFilters.java \
$(SRC)/filters/EdgeColorFilter.java \
$(SRC)/filters/FillColorFilter.java \
$(SRC)/filters/LineFilter.java \
$(SRC)/filters/ShapeFilter.java \
$(SRC)/utils/FlyweightFactory.java \
$(SRC)/utils/IconFactory.java \
$(SRC)/utils/IconItem.java \
$(SRC)/utils/PaintFactory.java \
$(SRC)/utils/StrokeFactory.java \
$(SRC)/utils/package-info.java \
$(SRC)/widgets/DrawingPanel.java \
$(SRC)/widgets/EditorFrame.java \
$(SRC)/widgets/InfoPanel.java \
$(SRC)/widgets/JLabeledComboBox.java \
$(SRC)/widgets/package-info.java

OTHER = $(SRC)/images/About.png \
$(SRC)/images/About_small.png \
$(SRC)/images/Black.png \
$(SRC)/images/Blue.png \
```



```
$(SRC)/images/Circle.png \
$(SRC)/images/Circle_small.png \
$(SRC)/images/ClearFilter.png \
$(SRC)/images/ClearFilter_small.png \
$(SRC)/images/Cyan.png \
$(SRC)/images/Dashed.png \
$(SRC)/images/Dashed_small.png \
$(SRC)/images/Delete.png \
$(SRC)/images/Delete_small.png \
$(SRC)/images/EdgeColor.png \
$(SRC)/images/EdgeColor_small.png \
$(SRC)/images/Ellipse.png \
$(SRC)/images/Ellipse_small.png \
$(SRC)/images/FillColor.png \
$(SRC)/images/FillColor_small.png \
$(SRC)/images/Filter.png \
$(SRC)/images/Filter_small.png \
$(SRC)/images/Green.png \
$(SRC)/images/Logo.png \
$(SRC)/images/Magenta.png \
$(SRC)/images/None.png \
$(SRC)/images/None_small.png \
$(SRC)/images/Orange.png \
$(SRC)/images/Others.png \
$(SRC)/images/Polygon.png \
$(SRC)/images/Polygon_small.png \
$(SRC)/images/Quit.png \
$(SRC)/images/Quit_small.png \
$(SRC)/images/Rectangle.png \
$(SRC)/images/Rectangle_small.png \
$(SRC)/images/Red.png \
"$(SRC)/images/Rounded Rectangle.png" \
"$(SRC)/images/Rounded Rectangle_small.png" \
$(SRC)/images/Solid.png \
$(SRC)/images/Solid_small.png \
$(SRC)/images/Undo.png \
$(SRC)/images/Undo_small.png \
$(SRC)/images/White.png \
$(SRC)/images/Yellow.png \
TP5.pdf

.PHONY : doc ps

# Les targets de compilation
# pour générer l'application
all : $(foreach name, $(MAIN), $(BIN)/$(name).class)

#règle de compilation générique
$(BIN)/%.class : $(SRC)/%.java
	$(JAVAC) -sourcepath $(SRC) -classpath $(BIN):$(CLASSPATH) -d $(BIN) $(CFLAGS) $<

# Edition des sources $(EDITOR) doit être une variable d'environnement
edit :
	$(EDITOR) $(SOURCES) Makefile &

# nettoyer le répertoire
clean :
	find bin/ -type f -name "*.class" -exec rm -f {} \;
	rm -rf *~ $(DOC)/* $(LISTDIR)/*

#realclean : clean
#	rm -f $(ARCHDIR)/*.$(ARCHFMT)

# générer le listing
$(LISTDIR) :
	mkdir $(LISTDIR)

ps : $(LISTDIR)
	$(A2PS) -2 --file-align=fill --line-numbers=1 --font-size=10 \
	--chars-per-line=100 --tabsize=4 --pretty-print \
	--highlight-level=heavy --prologue="gray" \
	-o$(LISTDIR)/$(OUTPUT).ps Makefile $(SOURCES)

pdf : ps
	$(PS2PDF) $(LISTDIR)/$(OUTPUT).ps $(LISTDIR)/$(OUTPUT).pdf

# générer le listing lisible pour Gérard
bigps :
	$(A2PS) -1 --file-align=fill --line-numbers=1 --font-size=10 \
	--chars-per-line=100 --tabsize=4 --pretty-print \
	--highlight-level=heavy --prologue="gray" \
	-o$(LISTDIR)/$(OUTPUT).ps Makefile $(SOURCES)
```

```
165
166 bigpdf : bigps
167     $(PS2PDF) $(LISTDIR)/$(OUTPUT).ps $(LISTDIR)/$(OUTPUT).pdf
168
169 # voir le listing
170 preview : ps
171     $(GHOSTVIEW) $(LISTDIR)/$(OUTPUT); rm -f $(LISTDIR)/$(OUTPUT) $(LISTDIR)/$(OUTPUT)~
172
173 # générer la doc avec javadoc
174 doc : $(SOURCES)
175     $(DOCP) -private -d $(DOC) -author -link $(LOCALLINK) $(SOURCES)
176 #   $(DOCP) -private -d $(DOC) -author -linkoffline $(WEBLINK) $(LOCALLINK) $(SOURCES)
177
178 # générer une archive de sauvegarde
179 $(ARCHDIR) :
180     mkdir $(ARCHDIR)
181
182 archive : pdf $(ARCHDIR)
183     $(ARCH) $(ARCHDIR)/$(ARCHIVE)-$(DATE).$(ARCHFMT) $(SOURCES) $(LISTDIR)/*.pdf $(OTHER) $(BIN) Mak
    efile $(FIGDIR)/*.pdf
184
185 # exécution des programmes de test
186 run : all
187     $(foreach name, $(MAIN), $(JAVA) -classpath $(BIN):$(CLASSPATH) $(name) $(JAVAOPTIONS) )
```



```
1  import java.awt.EventQueue;
2
3  import javax.swing.UIManager;
4  import javax.swing.UnsupportedLookAndFeelException;
5
6  import widgets.EditorFrame;
7
8  /**
9   * Programme principal lançant la fenêtre {@link EditorFrame}
10  * @author davidroussel
11  */
12 public class Editor
13 {
14
15     /**
16      * Programme principal
17      * @param args arguments [non utilisés]
18      */
19     public static void main(String[] args)
20     {
21
22         /*
23          * Mise ne place du look and feel du système
24          */
25         try
26         {
27             UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
28         }
29         catch (ClassNotFoundException e)
30         {
31             e.printStackTrace();
32         }
33         catch (InstantiationException e)
34         {
35             e.printStackTrace();
36         }
37         catch (IllegalAccessException e)
38         {
39             e.printStackTrace();
40         }
41         catch (UnsupportedLookAndFeelException e)
42         {
43             e.printStackTrace();
44         }
45
46         // Mise en place spécifique à Mac OS X
47         String osName = System.getProperty("os.name");
48         if (osName.startsWith("Mac OS"))
49         {
50             macOSSettings();
51         }
52
53         /*
54          * Création de la fenêtre
55          */
56         final EditorFrame frame = new EditorFrame();
57
58         /*
59          * Insertion de la fenêtre dans la file des évènements GUI
60          */
61         EventQueue.invokeLater(new Runnable()
62         {
63             @Override
64             public void run()
65             {
66                 try
67                 {
68                     frame.pack();
69                     frame.setVisible(true);
70                 }
71                 catch (Exception e)
72                 {
73                     e.printStackTrace();
74                 }
75             }
76         });
77
78     }
79
80     /**
81      * Mise en place des options spécifiques à MacOS
82      * A virer si votre système n'est pas MacOS car com.apple.... risque
```

```
     * de ne pas exister
     */
    private static void macOSSettings()
    {
        // Remettre les menus au bon endroit (dans la barre en haut)
        System.setProperty("apple.laf.useScreenMenuBar", "true");

//      ImageIcon imageIcon = new ImageIcon(
//              Editor.class.getResource("/images/Logo.png"));
//      if (imageIcon.getImageLoadStatus() == MediaTracker.COMPLETE)
//      {
            // Titre de l'application
            System.setProperty(
                    "com.apple.mrj.application.apple.menu.about.name",
                    "Figure Editor");
//          // Chargement d'une icône pour le dock
//          com.apple.eawt.Application.getApplication().setDockIconImage(
//                  imageIcon.getImage());
//      }
    }
}

```



```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.GradientPaint;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.geom.Arc2D;
import java.awt.geom.Ellipse2D;
import java.awt.geom.GeneralPath;
import java.awt.geom.Line2D;
import java.awt.geom.Path2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.RoundRectangle2D;

import javax.swing.JApplet;
import javax.swing.JFrame;

/*
 * Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met: -
 * Redistributions of source code must retain the above copyright notice, this
 * list of conditions and the following disclaimer. - Redistributions in binary
 * form must reproduce the above copyright notice, this list of conditions and
 * the following disclaimer in the documentation and/or other materials provided
 * with the distribution. - Neither the name of Oracle or the names of its
 * contributors may be used to endorse or promote products derived from this
 * software without specific prior written permission. THIS SOFTWARE IS PROVIDED
 * BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
 * EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
 * INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/*
 * This is like the FontDemo applet in volume 1, except that it uses the Java 2D
 * APIs to define and render the graphics and text.
 */

public class ShapesDemo2D extends JApplet
{
    protected final static int maxCharHeight = 15;
    protected final static int minFontSize = 6;

    protected final static Color bg = Color.white;
    protected final static Color fg = Color.black;
    protected final static Color red = Color.red;
    protected final static Color white = Color.white;

    protected final static BasicStroke stroke = new BasicStroke(2.0f);
    protected final static BasicStroke wideStroke = new BasicStroke(8.0f,
            BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);

    protected final static float lastWidth = 20.0f;
    protected final static float dash1[] = { 2*lastWidth };
    protected final static BasicStroke dashed = new BasicStroke(1.0f,
            BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, 10.0f, dash1, 0.0f);
    protected final static BasicStroke fatDashed = new BasicStroke(lastWidth,
            BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, lastWidth, dash1, 0.0f);
    protected Dimension totalSize;
    protected FontMetrics fontMetrics;

    @Override
    public void init()
    {
        // Initialize drawing colors
        setBackground(bg);
        setForeground(fg);
    }

    FontMetrics pickFont(Graphics2D g2, String longString, int xSpace)
```

```
    {
        boolean fontFits = false;
        Font font = g2.getFont();
        FontMetrics fontMetrics = g2.getFontMetrics();
        int size = font.getSize();
        String name = font.getName();
        int style = font.getStyle();

        while (¬fontFits)
        {
            if ((fontMetrics.getHeight() ≤ maxCharHeight)
                    ∧ (fontMetrics.stringWidth(longString) ≤ xSpace))
            {
                fontFits = true;
            }
            else
            {
                if (size ≤ minFontSize)
                {
                    fontFits = true;
                }
                else
                {
                    g2.setFont(font = new Font(name, style, --size));
                    fontMetrics = g2.getFontMetrics();
                }
            }
        }

        return fontMetrics;
    }

    @Override
    public void paint(Graphics g)
    {
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        Dimension d = getSize();
        int gridWidth = d.width / 6;
        int gridHeight = d.height / 2;

        fontMetrics = pickFont(g2, "Filled and Stroked GeneralPath", gridWidth);

        Color fg3D = Color.lightGray;

        // on commence par effacer le fond
        g2.setColor(getBackground());
        g2.fillRect(0, 0, d.width, d.height);

        g2.setPaint(fg3D);
        g2.draw3DRect(0, 0, d.width - 1, d.height - 1, true);
        g2.draw3DRect(3, 3, d.width - 7, d.height - 7, false);
        g2.setPaint(fg);

        int x = 5;
        int y = 7;
        int rectWidth = gridWidth - (2 * x);
        int stringY = gridHeight - 3 - fontMetrics.getDescent();
        int rectHeight = stringY - fontMetrics.getMaxAscent() - y - 2;

        // draw Line2D.Double
        g2.draw(new Line2D.Double(x, (y + rectHeight) - 1, x + rectWidth, y));
        g2.drawString("Line2D", x, stringY);
        x += gridWidth;

        // draw Rectangle2D.Double
        g2.setStroke(stroke);
        g2.draw(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
        g2.drawString("Rectangle2D", x, stringY);
        x += gridWidth;

        // draw RoundRectangle2D.Double
        g2.setStroke(dashed);
        g2.draw(new RoundRectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10));
        g2.drawString("RoundRectangle2D", x, stringY);
        x += gridWidth;

        // draw Arc2D.Double
        g2.setStroke(wideStroke);
        g2.draw(new Arc2D.Double(x, y, rectWidth, rectHeight, 90, 135,
                Arc2D.OPEN));
```



```
        g2.drawString("Arc2D", x, stringY);
        x += gridWidth;

        // draw Ellipse2D.Double
        g2.setStroke(stroke);
        g2.draw(new Ellipse2D.Double(x, y, rectWidth, rectHeight));
        g2.drawString("Ellipse2D", x, stringY);
        x += gridWidth;

        // draw GeneralPath (polygon)
        int x1Points[] = { x, x + rectWidth, x, x + rectWidth };
        int y1Points[] = { y, y + rectHeight, y + rectHeight, y };
        GeneralPath polygon = new GeneralPath(Path2D.WIND_EVEN_ODD,
                x1Points.length);
        polygon.moveTo(x1Points[0], y1Points[0]);
        for (int index = 1; index < x1Points.length; index++)
        {
            polygon.lineTo(x1Points[index], y1Points[index]);
        }
        ;
        polygon.closePath();

        g2.draw(polygon);
        g2.drawString("GeneralPath", x, stringY);

        // NEW ROW
        x = 5;
        y += gridHeight;
        stringY += gridHeight;

        // draw GeneralPath (polyline)

        int x2Points[] = { x, x + rectWidth, x, x + rectWidth };
        int y2Points[] = { y, y + rectHeight, y + rectHeight, y };
        GeneralPath polyline = new GeneralPath(Path2D.WIND_EVEN_ODD,
                x2Points.length);
        polyline.moveTo(x2Points[0], y2Points[0]);
        for (int index = 1; index < x2Points.length; index++)
        {
            polyline.lineTo(x2Points[index], y2Points[index]);
        }

        g2.draw(polyline);
        g2.drawString("GeneralPath (open)", x, stringY);
        x += gridWidth;

        // fill Rectangle2D.Double (red)
        g2.setPaint(red);
        g2.fill(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
        g2.setPaint(fg);
        g2.drawString("Filled Rectangle2D", x, stringY);
        x += gridWidth;

        // fill RoundRectangle2D.Double
        GradientPaint redtowhite = new GradientPaint(x, y, red, x + rectWidth,
                y, white);
        g2.setPaint(redtowhite);
        g2.fill(new RoundRectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10));
        g2.setPaint(fg);
        g2.drawString("Filled RoundRectangle2D", x, stringY);
        x += gridWidth;

        // fill Arc2D
        g2.setPaint(red);
        g2.fill(new Arc2D.Double(x, y, rectWidth, rectHeight, 90, 135,
                Arc2D.OPEN));
        g2.setPaint(fg);
        g2.drawString("Filled Arc2D", x, stringY);
        x += gridWidth;

        // fill Ellipse2D.Double
        redtowhite = new GradientPaint(x, y, red, x + rectWidth, y, white);
        g2.setPaint(redtowhite);
        g2.fill(new Ellipse2D.Double(x, y, rectWidth, rectHeight));
        g2.setPaint(fg);
        g2.drawString("Filled Ellipse2D", x, stringY);
        x += gridWidth;

        // fill and stroke GeneralPath
        int x3Points[] = { x, x + rectWidth, x, x + rectWidth };
        int y3Points[] = { y, y + rectHeight, y + rectHeight, y };
        GeneralPath filledPolygon = new GeneralPath(Path2D.WIND_EVEN_ODD,
```

```
                x3Points.length);
        filledPolygon.moveTo(x3Points[0], y3Points[0]);
        for (int index = 1; index < x3Points.length; index++)
        {
            filledPolygon.lineTo(x3Points[index], y3Points[index]);
        }
        filledPolygon.closePath();

        g2.setPaint(red);
        g2.fill(filledPolygon);

        g2.setStroke(fatDashed);
        g2.setPaint(fg);
        g2.draw(filledPolygon);

        g2.drawString("Filled and Stroked GeneralPath", x, stringY);
    }

    public static void main(String s[])
    {
        JFrame f = new JFrame("ShapesDemo2D");
        f.addWindowListener(new WindowAdapter()
        {
            @Override
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
        JApplet applet = new ShapesDemo2D();
        f.getContentPane().add("Center", applet);
        applet.init();
        f.pack();
        f.setSize(new Dimension(550, 100));
        f.setVisible(true);
    }

}
```



```
package figures;

import java.awt.BasicStroke;
import java.awt.Graphics2D;
import java.awt.Paint;
import java.awt.Shape;
import java.awt.geom.Point2D;
import java.awt.geom.Rectangle2D;

import figures.enums.FigureType;

/**
 * Classe commune à toutes les sortes de figures
 *
 * @author davidroussel
 */
public abstract class Figure
{
    /**
     * La forme à dessiner
     */
    protected Shape shape;

    /**
     * Couleur du bord de la figure
     */
    protected Paint edge;

    /**
     * Couleur de remplissage de la figure
     */
    protected Paint fill;

    /**
     * Caractéristiques de la bordure des figure : épaisseur, forme des
     * extremités et [evt] forme des jointures
     */
    protected BasicStroke stroke;

    /**
     * Le numéro d'instance de cette figure.
     * 1 si c'est la première figure de ce type, etc.
     */
    protected int instanceNumber;

    /**
     * Constructeur d'une figure abstraite à partir d'un style de ligne d'une
     * couleur de bordure et d'une couleur de remplissage. Les styles de lignes
     * et les couleurs étant souvent les même entre les différentes figures ils
     * devront être fournis par un flyweight. Le stroke, le edge et le fill
     * peuvent chacun être null.
     *
     * @param stroke caractéristiques de la ligne de bordure
     * @param edge couleur de la ligne de bordure
     * @param fill couleur (ou gradient de couleurs) de remplissage
     */
    protected Figure(BasicStroke stroke, Paint edge, Paint fill)
    {
        this.stroke = stroke;
        this.edge = edge;
        this.fill = fill;
        shape = null;
    }

    /**
     * Déplacement du dernier point de la figure (utilisé lors du dessin d'une
     * figure tant que l'on déplace le dernier point)
     *
     * @param p la nouvelle position du dernier point
     */
    public abstract void setLastPoint(Point2D p);

    /**
     * Dessin de la figure dans un contexte graphique fournit par le système.
     * Met en place le stroke et les couleur, puis dessine la forme géométrique
     * correspondant à la figure (figure remplie d'abord si le fill est non
     * null, puis bordure si le edge est non null)
     *
     * @param g2D le contexte graphique
     */
    public final void draw(Graphics2D g2D)
    {
```

```
83          if (fill ≠ null)
84          {
85              g2D.setPaint(fill);
86              g2D.fill(shape);
87          }
88          if ((edge ≠ null) ∧ (stroke ≠ null))
89          {
90              g2D.setStroke(stroke);
91              g2D.setPaint(edge);
92              g2D.draw(shape);
93          }
94      }
95
96      /**
97       * Obtention du nom de la figure. Le nom d'une figure est composé de son
98       * type suivi par le numéro de l'instance de ce type
99       *
100      * @return le nom de la figure
101      */
102     public String getName()
103     {
104         return new String(getClass().getSimpleName() + "" + instanceNumber);
105     }
106
107     /**
108      * Obtention du rectangle englobant de la figure.
109      * Obtenu grâce au {@link Shape#getBounds2D()}
110      * @return le rectangle englobant de la figure
111      */
112     public Rectangle2D getBounds2D()
113     {
114         return shape.getBounds2D();
115     }
116
117     /**
118      * Obtention du barycentre de la figure.
119      * @return le point correspondant au barycentre de la figure
120      */
121     public abstract Point2D getCenter();
122
123     /**
124      * Teste si le point p est contenu dans cette figure.
125      * Utilise {@link Shape#contains(Point2D)}
126      * @param p le point dont on veut tester s'il est contenu dans la figure
127      * @return true si le point p est contenu dans la figure, false sinon
128      */
129     public boolean contains(Point2D p)
130     {
131         return shape.contains(p);
132     }
133
134     /**
135      * Accesseur du type de figure selon {@link FigureType}
136      * @return le type de figure
137      */
138     public abstract FigureType getType();
139
140     /**
141      * Accesseur en lecture de la forme interne
142      * @return la forme interne
143      */
144     public Shape getShape()
145     {
146         return shape;
147     }
148
149     /**
150      * Accesseur en lecture du {@link Paint} du contour
151      * @return le {@link Paint} du contour
152      */
153     public Paint getEdgePaint()
154     {
155         return edge;
156     }
157
158     /**
159      * Accesseur en lecture du {@link Paint} du remplissage
160      * @return le {@link Paint} du remplissage
161      */
162     public Paint getFillPaint()
163     {
164         return fill;
```



```
165     }
166
167     /**
168      * Accesseur en lecture du {@link BasicStroke} du contour
169      * @return le {@link BasicStroke} du contour
170      */
171     public BasicStroke getStroke()
172     {
173         return stroke;
174     }
175 }
```

```java
package figures;

import java.awt.BasicStroke;
import java.awt.Paint;
import java.awt.geom.Point2D;
import java.util.Observable;
import java.util.Observer;
import java.util.Vector;
import java.util.stream.Stream;

import figures.enums.FigureType;
import figures.enums.LineType;
import filters.EdgeColorFilter;
import filters.FigureFilters;
import filters.FillColorFilter;
import filters.LineFilter;
import filters.ShapeFilter;
import utils.PaintFactory;
import utils.StrokeFactory;

/**
 * Classe contenant l'ensemble des figures à dessiner (LE MODELE)
 *
 * @author davidroussel
 */
public class Drawing extends Observable
{
    /**
     * Liste des figures à dessiner
     */
    private Vector<Figure> figures;

    /**
     * Le type de figure à créer
     */
    private FigureType type;

    /**
     * La couleur de remplissage courante
     */
    private Paint fillPaint;

    /**
     * La couleur de trait courante
     */
    private Paint edgePaint;

    /**
     * La largeur de trait courante
     */
    private float edgeWidth;

    /**
     * Le type de trait courant (sans trait, trait plein, trait pointillé)
     */
    private LineType edgeType;

    /**
     * Les caractétistique à appliquer au trait en fonction de {@link #type} et
     * {@link #edgeWidth}
     */
    private BasicStroke stroke;

    /**
     * Figure située sous le curseur.
     * Déterminé par {@link #getFigureAt(Point2D)}
     */
    private Figure selectedFigure;

    /**
     * Etat de filtrage des figures dans le flux de figures fournit par
     * {@link #stream()}
     * Lorsque {@link #filtering} est true le dessin des figures est filtré
     * par l'ensemble des filtres présents dans {@link #shapeFilters},
     * {@link #fillColorFilter}, {@link #edgeColorFilter} et {@link #lineFilters}.
     * Lorsque {@link #filtering} est false, toutes les figures sont fournies
     * dans le flux des figures.
     * @see #stream()
     */
    private boolean filtering;

    /**
```



```java
     * Filtres à appliquer au flux des figures pour sélectionner les types
     * de figures à afficher
     * @see #stream()
     */
    private FigureFilters<FigureType> shapeFilters;

    /**
     * Filtre à appliquer au flux des figures pour sélectionner les figures
     * ayant une couleur particulière de remplissage
     */
    private FillColorFilter fillColorFilter;

    /**
     * Filtre à appliquer au flux des figures pour sélectionner les figures
     * ayant une couleur particulière de trait
     */
    private EdgeColorFilter edgeColorFilter;

    /**
     * Filtres à appliqué au flux des figures pour sélectionner les figures
     * ayant un type particulier de lignes
     */
    private FigureFilters<LineType> lineFilters;

    /**
     * Constructeur de modèle de dessin
     */
    public Drawing()
    {
        figures = new Vector<Figure>();
        shapeFilters = new FigureFilters<FigureType>();

        fillColorFilter = null;
        edgeColorFilter = null;
        lineFilters = new FigureFilters<LineType>();

        fillPaint = null;
        edgePaint = null;
        edgeWidth = 1.0f;
        edgeType = LineType.SOLID;
        stroke = StrokeFactory.getStroke(edgeType, edgeWidth);
        filtering = false;
        selectedFigure = null;

        System.out.println("Drawing model created");
    }

    /**
     * Nettoyage avant destruction
     */
    @Override
    protected void finalize()
    {
        // Aide au GC
        figures.clear();
    }

    /**
     * Mise à jour du ou des {@link Observer} qui observent ce modèle. On place
     * le modèle dans un état "changé" puis on notifie les observateurs.
     */
    public void update()
    {
        setChanged();
        notifyObservers();
    }

    // ------------------------------------------------------------------------
    // Accesseur et Mutateurs des attributs
    // ------------------------------------------------------------------------
    /**
     * Accesseur du type courant de figure
     * @return le type courant de figures à créer
     */
    public FigureType getType()
    {
        return type;
    }
    /**
     * Mise en place d'un nouveau type de figure à générer
     *
     * @param type le nouveau type de figure
```

```
165         */
166        public void setType(FigureType type)
167        {
168            this.type = type;
169        }
170
171        /**
172         * Accesseur de la couleur de remplissage courante des figures
173         * @return la couleur de remplissage courante des figures
174         */
175        public Paint getFillpaint()
176        {
177            return fillPaint;
178        }
179
180        /**
181         * Mise en place d'une nouvelle couleur de remplissage
182         *
183         * @param fillPaint la nouvelle couleur de remplissage
184         */
185        public void setFillPaint(Paint fillPaint)
186        {
187            this.fillPaint = fillPaint;
188            /*
189             * Au moment où on initiera une nouvelle figure, on mettra ce paint dans
190             * la PaintFactory
191             */
192        }
193
194        /**
195         * Accesseur de la couleur de trait courante des figures
196         * @return la couleur de remplissage courante des figures
197         */
198        public Paint getEdgePaint()
199        {
200            return edgePaint;
201        }
202
203        /**
204         * Mise en place d'une nouvelle couleur de trait
205         *
206         * @param edgePaint la nouvelle couleur de trait
207         */
208        public void setEdgePaint(Paint edgePaint)
209        {
210            this.edgePaint = edgePaint;
211            /*
212             * Au moment où on initiera une nouvelle figure, on mettra ce paint dans
213             * la PaintFactory
214             */
215        }
216
217        /**
218         * Mise en place d'un nouvelle épaisseur de trait
219         *
220         * @param width la nouvelle épaisseur de trait
221         */
222        public void setEdgeWidth(float width)
223        {
224            edgeWidth = width;
225            /*
226             * Au moment où on initiera une nouvelle figure, on mettra le stroke
227             * résultant dans la StrokeFactory
228             */
229        }
230
231        /**
232         * Mise en place d'un nouvel état de ligne pointillée
233         *
234         * @param type le nouveau type de ligne
235         */
236        public void setEdgeType(LineType type)
237        {
238            edgeType = type;
239            /*
240             * Au moment où on initiera une nouvelle figure, on mettra le stroke
241             * résultant dans la StrokeFactory
242             */
243        }
244
245        /**
246         * Initialisation d'une figure de type {@link #type} au point p et ajout de
```



```
247         * cette figure à la liste des {@link #figures}
248         *
249         * @param p le point où initialiser la figure
250         * @return la nouvelle figure créée à x et y avec les paramètres courants
251         */
252        public Figure initiateFigure(Point2D p)
253        {
254            /*
255             * Maintenant que l'on s'apprête effectivement à créer une figure on
256             * ajoute les Paints et le Stroke aux factories
257             */
258            fillPaint = PaintFactory.getPaint(fillPaint);
259            edgePaint = PaintFactory.getPaint(edgePaint);
260            stroke = StrokeFactory.getStroke(edgeType, edgeWidth);
261
262            /*
263             * Obtention de la figure correspondant au type de figure choisi grâce à
264             * type.getFigure(...)
265             */
266            Figure newFigure = type.getFigure(stroke,
267                                              edgePaint,
268                                              fillPaint,
269                                              p);
270
271            /*
272             * Ajout de la figure à #figures
273             */
274            if (newFigure ≠ null)
275            {
276                figures.add(newFigure);
277            }
278
279            /* Notification des observers */
280            update();
281
282            return newFigure;
283        }
284
285        /**
286         * Obtention de la dernière figure (implicitement celle qui est en cours de
287         * dessin)
288         * @return la dernière figure du dessin
289         */
290        public Figure getLastFigure()
291        {
292            if (¬figures.isEmpty())
293            {
294                return figures.get(figures.size() − 1);
295            }
296            else
297            {
298                System.err.println("Drawing.getLastFigure : empty");
299                return null;
300            }
301        }
302
303        /**
304         * Obtention de la dernière figure contenant le point p.
305         * @param p le point sous lequel on cherche une figure
306         * @return une référence vers la dernière figure contenant le point p ou à
307         *          défaut null.
308         */
309        public Figure getFigureAt(Point2D p)
310        {
311            // Première version du parcours sans itérateur
312    //      for (int i = figures.size() − 1; i >= 0; i−−)
313    //      {
314    //          AbstractFigure figure = figures.get(i);
315    //          if (figure.contains(p))
316    //          {
317    //              return figure;
318    //          }
319    //      }
320
321    //      // Seconde version du parcours avec le ListIterator initialisé à listIterator(figures.size()
       )
322    //      for (ListIterator<Figure> it = reverseIterator(); it.hasPrevious();)
323    //      {
324    //          Figure figure = it.previous();
325    //          if (figure.contains(p))
326    //          {
327    //              return figure;
```

```
//            }
//        }

        /*
         * Recherche dans le flux des figures de la DERNIERE figure contenant
         * le point p
         */
        selectedFigure = null;

        stream().forEach((Figure f) →
        {
            if (f.contains(p))
            {
                selectedFigure = f;
            }
        });

        return selectedFigure;
    }

    /**
     * Retrait de la dernière figure (sera déclencé par une action undo)
     * @post le modèle de dessin a été mis à jour
     */
    public void removeLastFigure()
    {
        if (¬figures.isEmpty())
        {
            figures.remove(figures.size() − 1);
            update();
        }
    }

    /**
     * Effacement de toutes les figures (sera déclenché par une action clear)
     * @post le modèle de dessin a été mis à jour
     */
    public void clear()
    {
        if (¬figures.isEmpty())
        {
            figures.clear();
            update();
        }
    }

    /**
     * Accesseur de l'état de filtrage
     * @return l'état courant de filtrage
     */
    public boolean getFiltering()
    {
        return filtering;
    }

    /**
     * Changement d'état du filtrage
     * @param filtering le nouveau statut de filtrage
     * @post le modèle de dessin a été mis à jour
     */
    public void setFiltering(boolean filtering)
    {
        this.filtering = filtering;
        update();
    }

    /**
     * Ajout d'un filtre pour filtrer les types de figures
     * @param filter le filtre à ajouter
     * @return true si le filtre n'était pas déjà présent dans l'ensemble des
     * filtres fitrant les types de figures, false sinon
     * @post si le filtre a été ajouté, une mise à jour est déclenchée
     */
    public boolean addShapeFilter(ShapeFilter filter)
    {
        boolean added = shapeFilters.add(filter);

        // System.out.println(shapeFilters);

        if (added)
        {
            update();
```





```
        }

        return added;
    }

    /**
     * Retrait d'un filtre filtrant les types de figures
     * @param filter le filtre à retirer
     * @return true si le filtre faisait partie des filtres filtrant les types
     * de figure et a été retiré, false sinon.
     * @post si le filtre a éré retiré, une mise à jour est déclenchée
     */
    public boolean removeShapeFilter(ShapeFilter filter)
    {
        boolean removed = shapeFilters.remove(filter);

        // System.out.println(shapeFilters);

        if (removed)
        {
            update();
        }

        return removed;
    }

    /**
     * Mise en place du filtre de couleur de remplissage
     * @param filter le filtre de couleur de remplissage à appliquer
     * @post le {@link #fillColorFilter} est mis en place et une mise à jour
     * est déclenchée
     */
    public void setFillColorFilter(FillColorFilter filter)
    {
        fillColorFilter = filter;
        update();
    }

    /**
     * Mise en place du filtre de couleur de trait
     * @param filter le filtre de couleur de trait à appliquer
     * @post le {@link #edgeColorFilter} est mis en place et une mise à jour
     * est déclenchée
     */
    public void setEdgeColorFilter(EdgeColorFilter filter)
    {
        edgeColorFilter = filter;
        update();
    }

    /**
     * Ajout d'un filtre pour filtrer les types de ligne des figures
     * @param filter le filtre à ajouter
     * @return true si le filtre n'était pas déjà présent dans l'ensemble des
     * filtres fitrant les types de lignes, false sinon
     * @post si le filtre a été ajouté, une mise à jour est déclenchée
     */
    public boolean addLineFilter(LineFilter filter)
    {
        boolean added = lineFilters.add(filter);

        if (added)
        {
            update();
        }

        return added;
    }

    /**
     * Retrait d'un filtre filtrant les types de lignes
     * @param filter le filtre à retirer
     * @return true si le filtre faisait partie des filtres filtrant les types
     * de lignes et a été retiré, false sinon.
     * @post si le filtre a éré retiré, une mise à jour est déclenchée
     */
    public boolean removeLineFilter(LineFilter filter)
    {
        boolean removed = lineFilters.remove(filter);

        if (removed)
        {
```

```
492                 update();
493             }
494
495             return removed;
496         }
497
498         /**
499          * Accès aux figures dans un stream afin que l'on puisse y appliquer
500          * de filtres
501          * @return le flux des figures éventuellement filtrés par les différents
502          * filtres
503          */
504         public Stream<Figure> stream()
505         {
506             Stream<Figure> figuresStream = figures.stream();
507             if (filtering)
508             {
509                 if (shapeFilters.size() > 0)
510                 {
511                     figuresStream = figuresStream.filter(shapeFilters);
512                 }
513
514                 if (fillColorFilter ≠ null)
515                 {
516                     figuresStream = figuresStream.filter(fillColorFilter);
517                 }
518
519                 if (edgeColorFilter ≠ null)
520                 {
521                     figuresStream = figuresStream.filter(edgeColorFilter);
522                 }
523
524                 if (lineFilters.size() > 0)
525                 {
526                     figuresStream = figuresStream.filter(lineFilters);
527                 }
528             }
529
530             return figuresStream;
531         }
532 }
```



```
1   package figures;
2
3   import java.awt.BasicStroke;
4   import java.awt.Paint;
5   import java.awt.geom.Ellipse2D;
6   import java.awt.geom.Point2D;
7
8   import figures.enums.FigureType;
9
10  /**
11   * Classe de Figure pour les cercles
12   *
13   * @author davidroussel
14   * @uml.dependency supplier="java.awt.geom.Ellipse2D.Float"
15   */
16  public class Circle extends Figure
17  {
18      /**
19       * Le rayon par défaut pour un cercle
20       */
21      public final static float DEFAULT_RAYON = 2.0f;
22
23      /**
24       * Le compteur d'instance des cercles. Utilisé pour donner un numéro
25       * d'instance après l'avoir incrémenté
26       */
27      private static int counter = 0;
28
29      /**
30       * Constructeur valué d'un cercle
31       *
32       * @param stroke le type du trait de la bordure
33       * @param edge la couleur de la bordure
34       * @param fill la couleur de remplissage
35       * @param center le centre du cercle
36       * @param rayon le rayon du cercle
37       */
38      public Circle(BasicStroke stroke, Paint edge, Paint fill, Point2D center,
39              float rayon)
40      {
41          super(stroke, edge, fill);
42          instanceNumber = ++counter;
43          float width = rayon * 2.0f;
44          float height = width;
45          float x = (float) (center.getX() - rayon);
46          float y = (float) (center.getY() - rayon);
47          shape = new Ellipse2D.Float(x, y, width, height);
48
49          // System.out.println("Cercle created");
50      }
51
52      /**
53       * Déplacement du dernier point de la ligne (utilisé lors du dessin d'un
54       * cercle pour faire varier le centre et le rayon tant que l'on déplace un
55       * point)
56       *
57       * @param p la nouvelle position du dernier point
58       * @see figures.Figure#setLastPoint(Point2D)
59       */
60      @Override
61      public void setLastPoint(Point2D p)
62      {
63          Ellipse2D.Float ellipse = (Ellipse2D.Float) shape;
64          float newWidth = (float) (p.getX() - ellipse.x);
65          float newHeight = (float) (p.getY() - ellipse.y);
66          float size = (Math.abs(newWidth) < Math.abs(newHeight) ? newWidth
67                  : newHeight);
68          ellipse.width = size;
69          ellipse.height = size;
70      }
71
72      /**
73       * Obtention du barycentre de la figure.
74       *
75       * @return le point correspondant au barycentre de la figure
76       */
77      @Override
78      public Point2D getCenter()
79      {
80          Ellipse2D.Float ellipse = (Ellipse2D.Float) shape;
81
82          return new Point2D.Float((float) ellipse.getCenterX(),
```

```
                (float) ellipse.getCenterY());
    }

    /**
     * Accesseur du type de figure selon {@link FigureType}
     * @return le type de figure
     */
    @Override
    public FigureType getType()
    {
        return FigureType.CIRCLE;
    }
}
```



```
package figures;

import java.awt.BasicStroke;
import java.awt.Paint;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Point2D;

import figures.enums.FigureType;

/**
 * Classe de Ellipse pour les {@link Figure}
 *
 * @author davidroussel
 * @uml.dependency supplier="java.awt.geom.Ellipse2D.Float"
 */
public class Ellipse extends Figure
{
    /**
     * Le compteur d'instance des ellipses. Utilisé pour donner un numéro
     * d'instance après l'avoir incrémenté
     */
    private static int counter = 0;

    /**
     * Création d'un ellipse avec les points en haut à gauche et en bas à droite
     *
     * @param stroke le type de trait
     * @param edge la couleur du trait
     * @param fill la couleur de remplissage
     * @param topLeft le point en haut à gauche
     * @param bottomRight le point en bas à droite
     */
    public Ellipse(BasicStroke stroke, Paint edge, Paint fill, Point2D topLeft,
            Point2D bottomRight)
    {
        super(stroke, edge, fill);
        instanceNumber = ++counter;
        float x = (float) topLeft.getX();
        float y = (float) topLeft.getY();
        float w = (float) (bottomRight.getX() - x);
        float h = (float) (bottomRight.getY() - y);
        shape = new Ellipse2D.Float(x, y, w, h);
    }

    /**
     * Déplacement du point inférieur droit de l'ellipse
     *
     * @param p le point où placer le dernier point (point inférieur droit)
     * @see figures.Figure#setLastPoint(Point2D)
     */
    @Override
    public void setLastPoint(Point2D p)
    {
        if (shape ≠ null)
        {
            Ellipse2D.Float ellipse = (Ellipse2D.Float) shape;
            float newWidth = (float) (p.getX() - ellipse.x);
            float newHeight = (float) (p.getY() - ellipse.y);
            ellipse.width = newWidth;
            ellipse.height = newHeight;
        }
    }

    /**
     * Obtention du barycentre de la figure.
     *
     * @return le point correspondant au barycentre de la figure
     */
    @Override
    public Point2D getCenter()
    {
        Ellipse2D.Float ellipse = (Ellipse2D.Float) shape;

        return new Point2D.Double(ellipse.getCenterX(), ellipse.getCenterY());
    }

    /**
     * Accesseur du type de figure selon {@link FigureType}
     * @return le type de figure
     */
    @Override
    public FigureType getType()
```

```
83      {
84          return FigureType.ELLIPSE;
85      }
86
87  }
```



```
1   package figures;
2
3   import java.awt.BasicStroke;
4   import java.awt.Paint;
5   import java.awt.geom.Point2D;
6   import java.awt.geom.Rectangle2D;
7   import java.awt.geom.RectangularShape;
8
9   import figures.enums.FigureType;
10
11  /**
12   * Classe de Rectangle pour les {@link Figure}
13   *
14   * @author davidroussel
15   */
16  public class Rectangle extends Figure
17  {
18      /**
19       * Le compteur d'instance des cercles.
20       * Utilisé pour donner un numéro d'instance après l'avoir incrémenté
21       */
22      private static int counter = 0;
23
24      /**
25       * Création d'un rectangle avec les points en haut à gauche et en bas à
26       * droite
27       *
28       * @param stroke le type de trait
29       * @param edge la couleur du trait
30       * @param fill la couleur de remplissage
31       * @param topLeft le point en haut à gauche
32       * @param bottomRight le point en bas à droite
33       */
34      public Rectangle(BasicStroke stroke, Paint edge, Paint fill, Point2D topLeft,
35              Point2D bottomRight)
36      {
37          super(stroke, edge, fill);
38          instanceNumber = ++counter;
39          float x = (float) topLeft.getX();
40          float y = (float) topLeft.getY();
41          float w = (float) (bottomRight.getX() - x);
42          float h = (float) (bottomRight.getY() - y);
43
44          shape = new Rectangle2D.Float(x, y, w, h);
45
46          // System.out.println("Rectangle created");
47      }
48
49      /**
50       * Création d'un rectangle sans points (utilisé dans les classes filles
51       * pour initialiser seulement les couleur et le style de trait sans
52       * initialiser {@link #shape}.
53       *
54       * @param stroke le type de trait
55       * @param edge la couleur du trait
56       * @param fill la couleur de remplissage
57       */
58      protected Rectangle(BasicStroke stroke, Paint edge, Paint fill)
59      {
60          super(stroke, edge, fill);
61
62          shape = null;
63      }
64
65      /**
66       * Déplacement du point en bas à droite du rectangle à la position
67       * du point p
68       *
69       * @param p la nouvelle position du dernier point
70       * @see figures.Figure#setLastPoint(Point2D)
71       */
72      @Override
73      public void setLastPoint(Point2D p)
74      {
75          if (shape ≠ null)
76          {
77              Rectangle2D.Float rect = (Rectangle2D.Float) shape;
78              float newWidth = (float) (p.getX() - rect.x);
79              float newHeight = (float) (p.getY() - rect.y);
80              rect.width = newWidth;
81              rect.height = newHeight;
82          }
```

```
    }

    /**
     * Obtention du barycentre de la figure.
     * @return le point correspondant au barycentre de la figure
     */
    @Override
    public Point2D getCenter()
    {
        RectangularShape rect = (RectangularShape) shape;

        return new Point2D.Double(rect.getCenterX(), rect.getCenterY());
    }

    /**
     * Accesseur du type de figure selon {@link FigureType}
     * @return le type de figure
     */
    @Override
    public FigureType getType()
    {
        return FigureType.RECTANGLE;
    }
}
```



```
package figures;

import java.awt.BasicStroke;
import java.awt.Paint;
import java.awt.geom.Point2D;
import java.awt.geom.RoundRectangle2D;

import figures.enums.FigureType;

/**
 * Figure correspondant aux rectangle à coins arrondis
 * @author davidroussel
 */
public class RoundedRectangle extends Rectangle
{
    /**
     * Le compteur d'instance des rectangles à coins arrondis.
     * Utilisé pour donner un numéro d'instance après l'avoir incrémenté
     */
    private static int counter = 0;

    /**
     * Constructeur d'un rectangle à coins arrondis
     * @param stroke le type de trait
     * @param edge la couleur du trait
     * @param fill la couleur de remplissage
     * @param topLeft le point en haut à gauche
     * @param bottomRight le point en bas à droite
     * @param arcSize la taille de l'arrondi des coins
     */
    public RoundedRectangle(BasicStroke stroke, Paint edge, Paint fill,
            Point2D topLeft, Point2D bottomRight, int arcSize)
    {
        super(stroke, edge, fill);
        instanceNumber = ++counter;
        float x = (float) topLeft.getX();
        float y = (float) topLeft.getY();
        float width = (float) (bottomRight.getX() - x);
        float height = (float) (bottomRight.getY() - y);
        float minDim = (width < height ? width : height) / 2.0f;
        float actualArcSize = (arcSize < minDim ? arcSize : minDim);
        shape = new RoundRectangle2D.Float(x, y, width, height, actualArcSize,
                actualArcSize);

        // System.out.println("Rounded Rectangle created");
    }

    /*
     * (non-Javadoc)
     * @see figures.AbstractFigure#setLastPoint(Point2D)
     */
    @Override
    public void setLastPoint(Point2D p)
    {
        RoundRectangle2D.Float rect = (RoundRectangle2D.Float) shape;
        rect.width = (float) (p.getX() - rect.x);;
        rect.height = (float) (p.getY() - rect.y);
    }

    /**
     * Mise en place de la taille de l'arc en fonction de la position
     * d'un point par rapport au coin inférieur droit
     * @param p le point déterminant la taille de l'arc
     */
    public void setArc(Point2D p)
    {
        RoundRectangle2D.Float rect = (RoundRectangle2D.Float)shape;

        double bottomRightX = rect.getMaxX();
        double bottomRightY = rect.getMaxY();
        double x = p.getX();
        double y = p.getY();

        if (x > bottomRightX)
        {
            if (y < bottomRightY)
            {
                rect.arcwidth = (float) (bottomRightY - y);
                rect.archeight = rect.arcwidth;
            }
            else
            {
```

```
                    rect.arcwidth = 0;
                    rect.archeight = 0;
                }
            }
            else // x <= bottomRightX
            {
                if (y > bottomRightY)
                {
                    rect.arcwidth = (float) (bottomRightX - x);
                    rect.archeight = rect.arcwidth;
                }
                else
                {
                    rect.arcwidth = 0;
                    rect.archeight = 0;
                }
            }
        }

        /**
         * Accesseur du type de figure selon {@link FigureType}
         * @return le type de figure
         */
        @Override
        public FigureType getType()
        {
            return FigureType.ROUNDED_RECTANGLE;
        }
}
```



```
package figures;

import java.awt.BasicStroke;
import java.awt.Paint;
import java.awt.Point;
import java.awt.geom.Point2D;

import figures.enums.FigureType;

/**
 * Une classe représentant les ligne polygonales composées de 2 ou + de points
 * @author davidroussel
 */
public class Polygon extends Figure
{
    /**
     * Le compteur d'instance des cercles.
     * Utilisé pour donner un numéro d'instance après l'avoir incrémenté
     */
    private static int counter = 0;

    /**
     * Constructeur valué d'une ligne polygonale à partir d'un style de ligne,
     * d'une couleur et des deux premiers point de la ligne
     * @param stroke le style de la ligne
     * @param edgeColor la couleur de la ligne
     * @param fillColor la couleur de remplissage
     * @param point1 le premier point de la ligne
     * @param point2 le second point de la ligne
     */
    public Polygon(BasicStroke stroke, Paint edgeColor, Paint fillColor,
        Point point1, Point point2)
    {
        super(stroke, edgeColor, fillColor);
        instanceNumber = ++counter;

        java.awt.Polygon poly = new java.awt.Polygon();
        poly.addPoint(point1.x, point1.y);
        poly.addPoint(point2.x, point2.y);
        shape = poly;
    }

    /**
     * Ajout d'un point au polygone
     * @param x l'abcisse du point à ajouter
     * @param y l'ordonnée du point à ajouter
     */
    public void addPoint(int x, int y)
    {
        java.awt.Polygon poly = (java.awt.Polygon) shape;
        poly.addPoint(x, y);
    }

    /**
     * Suppression du dernier point du polygone.
     * Uniquement s'il y en a plus d'un
     */
    public void removeLastPoint()
    {
        java.awt.Polygon poly = (java.awt.Polygon) shape;

        if (poly.npoints > 1)
        {
            // Sauvegarde des coords des points - le dernier
            int[] xs = new int[poly.npoints-1];
            int[] ys = new int[poly.npoints-1];
            for (int i = 0; i < xs.length; i++)
            {
                xs[i] = poly.xpoints[i];
                ys[i] = poly.ypoints[i];
            }

            // reset poly
            poly.reset();

            // Reajout des points sauvegardés
            for (int i = 0; i < xs.length; i++)
            {
                poly.addPoint(xs[i], ys[i]);
            }
        }
    }
```

```
83
84      /**
85       * Déplacement du dernier point du polygone
86       * @param p la position du dernier point
87       * @see lines.AbstractLine#setLastPoint(Point2D)
88       */
89      @Override
90      public void setLastPoint(Point2D p)
91      {
92          java.awt.Polygon poly = (java.awt.Polygon) shape;
93          int lastIndex = poly.npoints-1;
94          if (lastIndex ≥ 0)
95          {
96              poly.xpoints[lastIndex] = Double.valueOf(p.getX()).intValue();
97              poly.ypoints[lastIndex] = Double.valueOf(p.getY()).intValue();
98          }
99      }
100
101     /**
102      * Obtention du barycentre de la figure.
103      * @return le point correspondant au barycentre de la figure
104      */
105     @Override
106     public Point2D getCenter()
107     {
108         java.awt.Polygon poly = (java.awt.Polygon) shape;
109
110         float xm = 0.0f;
111         float ym = 0.0f;
112
113         if (poly.npoints > 0)
114         {
115             for (int i = 0; i < poly.npoints; i++)
116             {
117                 xm += poly.xpoints[i];
118                 ym += poly.ypoints[i];
119             }
120
121             xm /= poly.npoints;
122             ym /= poly.npoints;
123         }
124
125         return new Point2D.Float(xm, ym);
126     }
127
128     /**
129      * Accesseur du type de figure selon {@link FigureType}
130      * @return le type de figure
131      */
132     @Override
133     public FigureType getType()
134     {
135         return FigureType.POLYGON;
136     }
137 }
```



```
1   package figures.creationListeners;
2
3   import java.awt.event.MouseEvent;
4   import java.awt.event.MouseListener;
5   import java.awt.event.MouseMotionListener;
6   import java.awt.geom.Point2D;
7
8   import javax.swing.JLabel;
9
10  import figures.Figure;
11  import figures.Drawing;
12
13  /**
14   * Listener (incomplet) des évènements souris pour créer une figure. Chaque
15   * figure (Cercle, Ellipse, Rectangle, etc) est graphiquement construite par une
16   * suite de pressed/drag/release ou de clicks qui peut être différente pour
17   * chaque type de figure. Aussi les classes filles devront implémenter leur
18   * propre xxxCreationListener assurant la gestion de la création d'une nouvelle
19   * figure.
20   *
21   * @author davidroussel
22   */
23  public abstract class AbstractCreationListener implements MouseListener,
24          MouseMotionListener
25  {
26      /**
27       * Le drawing model à modifier par ce creationListener. Celui ci contient
28       * tous les élements nécessaires à la modification du dessin par les
29       * évènements souris.
30       */
31      protected Drawing drawingModel;
32
33      /**
34       * La figure en cours de dessin. Obtenue avec
35       * {@link Drawing#initiateFigure(java.awt.geom.Point2D)}. Evite d'avoir à
36       * appeler {@link Drawing#getLastFigure()} à chaque fois que la figure en
37       * cours de construction est modifiée.
38       */
39      protected Figure currentFigure;
40
41      /**
42       * Le label dans lequel afficher les instructions nécessaires à la
43       * complétion de la figure
44       */
45      protected JLabel tipLabel;
46
47      /**
48       * Le point de départ de la création de la figure. Utilisé pour comparer le
49       * point de départ et le point terminal pour élminier les figures de taille
50       * 0;
51       */
52      protected Point2D startPoint;
53
54      /**
55       * Le point terminal de la création de la figure. Utilisé pour comparer le
56       * point de départ et le point terminal pour élminier les figures de taille
57       * 0;
58       */
59      protected Point2D endPoint;
60
61      /**
62       * le conseil par défaut à afficher dans le {@link #tipLabel}
63       */
64      public static final String defaultTip = new String(
65              "Cliquez pour initier une figure");
66
67      /**
68       * Le tableau de chaines de caractères contenant les conseils à
69       * l'utilisateur pour chacune des étapes de la création. Par exemple [0] :
70       * cliquez et maintenez enfoncé pour initier la figure [1] : relâchez pour
71       * terminer la figure
72       */
73      protected String[] tips;
74
75      /**
76       * Le nombre d'étapes (typiquement click->drag->release) nécessaires à la
77       * création de la figure
78       */
79      protected final int nbSteps;
80
81      /**
82       * L'étape actuelle de création de la figure
```

```
83      */
84     protected int currentStep;
85
86     /**
87      * Constructeur protégé (destiné à être utilisé par les classes filles)
88      *
89      * @param model le modèle de dessin à modifier par ce creationListener
90      * @param infoLabel le label dans lequel afficher les conseils d'utilisation
91      * @param nbSteps le nombres d'étapes de création de la figure
92      */
93     protected AbstractCreationListener(Drawing model, JLabel infoLabel,
94             int nbSteps)
95     {
96         drawingModel = model;
97         currentFigure = null;
98         tipLabel = infoLabel;
99         this.nbSteps = nbSteps;
100        currentStep = 0;
101
102        // Allocation du nombres de conseils utilisateurs nécessaires
103        tips = new String[(nbSteps > 0 ? nbSteps : 0)];
104
105        if (drawingModel == null)
106        {
107            System.err.println("AbstractCreationListener caution null "
108                    + "drawing model");
109        }
110
111        if (tipLabel == null)
112        {
113            System.err.println("AbstractCreationListener caution null "
114                    + "tip label");
115        }
116    }
117
118 //  /**
119 //   * Mise en place du label dans lequel afficher les conseils d'utilisation
120 //   *
121 //   * @param label le label dans lequel afficher les conseils d'utilisation
122 //   */
123 //  public void setTipLabel(JLabel label)
124 //  {
125 //      tipLabel = label;
126 //  }
127
128    /**
129     * Initialisation de la création d'une nouvelle figure, détermine le point
130     * de départ de la figure ({@link #startPoint}), initie une nouvelle figure
131     * à la position de l'évènement ({@link Drawing#initiateFigure(Point2D)}),
132     * met à jour le dessin {@link Drawing#update()}, puis passe à l'étape
133     * suivante en mettant à jour les conseils utilisateurs (
134     * {@link #updateTip()}). Pour la plupart des figures la création commence
135     * par un appui sur le bouton gauche de la souris. A utiliser dans
136     * {@link MouseListener#mousePressed(MouseEvent)} ou bien dans
137     * {@link MouseListener#mouseClicked(MouseEvent)} suivant la figure à créer.
138     *
139     * @param e l'évènement souris à utiliser pour initier la création d'une
140     *            nouvelle figure à la position de cet évènement
141     */
142    public void startFigure(MouseEvent e)
143    {
144        startPoint = e.getPoint();
145        currentFigure = drawingModel.initiateFigure(e.getPoint());
146
147        nextStep();
148
149        drawingModel.update();
150    }
151
152    /**
153     * Terminaison de la création d'une figure, remet l'étape courante à 0,
154     * détermine la position du point de terminaison de la figure (
155     * {@link #endPoint}), vérifie que la figure ainsi terminée n'est pas de
156     * taille 0 ({@link #checkZeroSizeFigure()}), puis met à jour le dessin (
157     * {@link Drawing#update()}) et les conseils utilisateurs (
158     * {@link #updateTip()}). A utiliser dans un
159     * {@link MouseListener#mousePressed(MouseEvent)} ou bien dans un
160     * {@link MouseListener#mouseClicked(MouseEvent)} suivant la figure à créer.
161     *
162     * @param e l'évènement souris à utiliser lors de la terminaison d'un figure
163     */
164    public void endFigure(MouseEvent e)
```





```
165    {
166        // Remise à zéro de currentStep pour pouvoir réutiliser ce
167        // listener sur une autre figure
168        nextStep();
169
170        endPoint = e.getPoint();
171
172        checkZeroSizeFigure();
173
174        drawingModel.update();
175    }
176
177    /**
178     * Passage à l'étape suivante et mise à jours des conseils utilisateurs
179     * relatifs à l'étape suivante.
180     * Lorsque le passage à l'étape suivante dépasse le nombre d'étapes prévues
181     * l'étape courante est remise à 0.
182     * @see #currentStep
183     * @see #updateTip()
184     */
185    protected void nextStep()
186    {
187        if (currentStep < (nbSteps-1))
188        {
189            currentStep++;
190        }
191        else
192        {
193            currentStep = 0;
194        }
195
196        updateTip();
197    }
198
199    /**
200     * Mise à jour du conseil dans le {@link #tipLabel} en fonction de l'étape
201     * courante
202     */
203    protected void updateTip()
204    {
205        if (tipLabel != null)
206        {
207            tipLabel.setText(tips[currentStep]);
208        }
209    }
210
211    /**
212     * Contrôle de la taille de la figure créée à effectuer à la fin de la
213     * création afin d'éliminer les figures de taille 0;
214     * @see #startPoint
215     * @see #endPoint
216     */
217    protected void checkZeroSizeFigure()
218    {
219        if (startPoint.distance(endPoint) < 1.0)
220        {
221            drawingModel.removeLastFigure();
222            System.err.println("Removed zero sized figure");
223        }
224    }
225 }
```

```java
package figures.creationListeners;

import java.awt.event.MouseEvent;

import javax.swing.JLabel;

import figures.Figure;
import figures.Drawing;

/**
 * Listener permettant d'enchainer les actions souris pour créer des formes
 * rectangulaires comme des rectangles ou des ellipse (evt des cercles):
 * <ol>
 *  <li>bouton 1 pressé et maintenu enfoncé</li>
 *  <li>déplacement de la souris avec le bouton enfoncé</li>
 *  <li>relachement du bouton</li>
 * </ol>
 * @author davidroussel
 */
public class RectangularShapeCreationListener extends AbstractCreationListener
{
    /**
     * Constructeur d'un listener à deux étapes: pressed->drag->release pour
     * toutes les figures à caractère rectangulaire (Rectangle, Ellipse, evt
     * Cercle)
     *
     * @param model le modèle de dessin à modifier par ce creationListener
     * @param tipLabel le label dans lequel afficher les conseils utilisateur
     */
    public RectangularShapeCreationListener(Drawing model, JLabel tipLabel)
    {
        super(model, tipLabel, 2);

        tips[0] = new String("Cliquez et maintenez enfoncé pour initier la figure");
        tips[1] = new String("Relâchez pour terminer la figure");

        updateTip();

        System.out.println("RectangularShapeCreationListener created");
    }

    /**
     * Création d'une nouvelle figure rectangulaire de taille 0 au point de
     * l'évènement souris, si le bouton appuyé est le bouton gauche.
     *
     * @param e l'évènement souris
     * @see AbstractCreationListener#startFigure(MouseEvent)
     * @see java.awt.event.MouseListener#mousePressed(java.awt.event.MouseEvent)
     */
    @Override
    public void mousePressed(MouseEvent e)
    {
        if (e.getButton() == MouseEvent.BUTTON1)
        {
            startFigure(e);
        }
    }

    /**
     * Terminaison de la nouvelle figure rectangulaire si le bouton appuyé
     * était le bouton gauche
     * @param e l'évènement souris
     * @see AbstractCreationListener#endFigure(MouseEvent)
     * @see java.awt.event.MouseListener#mouseReleased(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseReleased(MouseEvent e)
    {
        if (e.getButton() == MouseEvent.BUTTON1)
        {
            endFigure(e);
        }
    }

    /* (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseClicked(MouseEvent e)
    {
        // Rien
    }
```



```java

    /* (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseEntered(MouseEvent e)
    {
        // Rien
    }

    /* (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseExited(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseExited(MouseEvent e)
    {
        // Rien
    }

    /*
     * (non-Javadoc)
     * @see java.awt.event.MouseMotionListener#mouseMoved(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseMoved(MouseEvent e)
    {
        // Rien
    }

    /**
     * Déplacement du point en bas à droite de la figure rectangulaire, si
     * l'on se trouve à l'étape 1 (après initalisation de la figure) et que
     * le bouton enfoncé est bien le bouton gauche.
     * @see java.awt.event.MouseMotionListener#mouseDragged(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseDragged(MouseEvent e)
    {
        if (currentStep == 1)
        {
            // AbstractFigure figure = drawingModel.getLastFigure();
            Figure figure = currentFigure;
            if (figure != null)
            {
                figure.setLastPoint(e.getPoint());
            }
            drawingModel.update();
        }
    }
}
```

```java
package figures.creationListeners;

import java.awt.event.MouseEvent;

import javax.swing.JLabel;

import figures.Drawing;
import figures.RoundedRectangle;

/**
 * Listener permettant d'enchainer les actions souris nécessaires à la création
 * d'un Rectangle arrondi:
 *
 * @author davidroussel
 */
public class RoundedRectangleCreationListener extends AbstractCreationListener
{

    /**
     * Constructeur d'un Listener pour créer un polygon en plusieurs clicks
     *
     * @param model le modèle de dessin à modifier
     * @param infoLabel le label dans lequel afficher les consseils utilisateurs
     */
    public RoundedRectangleCreationListener(Drawing model, JLabel infoLabel)
    {
        super(model, infoLabel, 3);

        tips[0] = new String("Bouton gauche + drag pour commencer le regtangle");
        tips[1] = new String("Relâchez pour terminer le rectangle");
        tips[2] = new String("Clic gauche pour terminer l'arrondi du rectangle");

        updateTip();

        System.out.println("RoundedRectangleCreationListener created");
    }

    /**
     * Initiation de la création d'un rectangle arrondi et passage à l'étape
     * suivante
     *
     * @param e l'évènement souris associé
     * @see figures.creationListeners.AbstractCreationListener#mousePressed(java.awt.event.MouseEvent)
     */
    @Override
    public void mousePressed(MouseEvent e)
    {
        if ((e.getButton() == MouseEvent.BUTTON1) && (currentStep == 0))
        {
            startFigure(e);
        }
    }

    /**
     * Terminaison de la partie rectangle du rectangle arrondi et passage à
     * l'étape suivante
     *
     * @param e l'évènement souris associé
     * @see figures.creationListeners.AbstractCreationListener#mouseReleased(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseReleased(MouseEvent e)
    {
        // Terminaison du rectangle mais pas encore de l'arrondi
        if ((e.getButton() == MouseEvent.BUTTON1) && (currentStep == 1))
        {
            nextStep();
        }
    }

    /**
     * Après la partie terminaison de la partie rectangle, terminaison de la
     * partie arc (arrondi)
     *
     * @param e l'évènement souris associé
     * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseClicked(MouseEvent e)
    {
        if ((e.getButton() == MouseEvent.BUTTON1) && (currentStep == 2))
        {
            endFigure(e);
        }
    }

    /*
     * (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseEntered(MouseEvent e)
    {
        // Rien
    }

    /*
     * (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseExited(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseExited(MouseEvent e)
    {
        // Rien
    }

    /**
     * Modification de l'arrondi en fonction de la position du point de
     * l'évènement par rapport au coin inférieur droit du rectangle lorsque l'on
     * se trouve dans l'étape de modification de l'arrondi
     *
     * @param e l'évènement souris associé
     * @see java.awt.event.MouseMotionListener#mouseMoved(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseMoved(MouseEvent e)
    {
        if (currentStep == 2)
        {
            RoundedRectangle rect = (RoundedRectangle) currentFigure;
            rect.setArc(e.getPoint());

            drawingModel.update();
        }
    }

    /**
     * Déplacement du point en bas à droite du rectangle lorsque l'on est dans
     * l'étape de formation du rectangle
     *
     * @param e l'évènement souris associé
     * @see java.awt.event.MouseMotionListener#mouseDragged(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseDragged(MouseEvent e)
    {
        if (currentStep == 1)
        {
            // déplacement du coin inférieur droit du rectangle
            currentFigure.setLastPoint(e.getPoint());
            drawingModel.update();
        }
    }
}
```

```java
package figures.creationListeners;

import java.awt.Point;
import java.awt.event.MouseEvent;

import javax.swing.JLabel;

import figures.Figure;
import figures.Drawing;
import figures.Polygon;

/**
 * Listener permettant d'enchainer les actions souris nécessaires à la création
 * d'un polygone :
 * <ul>
 * <li>premier click avec le bouton gauche pour initier la création du polygone</li>
 * <li>les clicks suivants:
 * <ul>
 * <li>avec le bouton gauche ajoute un point au polygone</li>
 * <li>avec le bouton droit termine le polygone</li>
 * <li>avec le bouton du milieu retire le dernier point du polygone</li>
 * </ul>
 * </li>
 * <li>Une fois le polygone en cours de création, le déplacement de la souris
 * déplace le dernier point du polygone</li>
 * </ul>
 *
 * @author davidroussel
 */
public class PolygonCreationListener extends AbstractCreationListener
{

    /**
     * Constructeur d'un Listener pour créer un polygon en plusieurs clicks
     *
     * @param model le modèle de dessin à modifier
     * @param infoLabel le label dans lequel afficher les conseils utilisateurs
     */
    public PolygonCreationListener(Drawing model, JLabel infoLabel)
    {
        super(model, infoLabel, 2);

        tips[0] = new String("Clic gauche pour commencer le polygone");
        tips[1] = new String("clic gauche pour ajouter / droit pour terminer");

        updateTip();

        System.out.println("PolygonCreationListener created");
    }

    /*
     * (non-Javadoc)
     * @see
     * figures.creationListeners.AbstractCreationListener#mousePressed(java.
     * awt.event.MouseEvent)
     */
    @Override
    public void mousePressed(MouseEvent e)
    {
        // Rien
    }

    /*
     * (non-Javadoc)
     * @see
     * figures.creationListeners.AbstractCreationListener#mouseReleased(java
     * .awt.event.MouseEvent)
     */
    @Override
    public void mouseReleased(MouseEvent e)
    {
        // Rien
    }

    /**
     * Actions à réaliser lorsqu'un bouton de la souris est cliqué.
     * Si l'on se trouve à l'étape 0 et que le bouton cliqué est
     * {@link MouseEvent#BUTTON1}, on initie la figure et on passe à l'étape suivante.
     * Dans l'étape suivante si c'est {@link MouseEvent#BUTTON1} qui est cliqué
     * on ajoute un point, si c'est le {@link MouseEvent#BUTTON2} on supprime le
     * dernier point ajouté et si c'est le bouton {@link MouseEvent#BUTTON3},
     * on termine la figure.
     * @param e l'évènement souris associé
     * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseClicked(MouseEvent e)
    {
        Point p = e.getPoint();
        /*
         * Initie la création d'un premier point fixé à l'endroit du click
         * puis d'un deuxième point (créé au même endroit) qui se déplacera avec
         * le pointeur de la souris. un nouveau click fixera ce nouveau point et
         * en ajoutera un autre lui aussi attaché au pointeur de la souris, et
         * ainsi de suite. Le dernier point est retiré si l'utilisateur clique
         * avec le bouton du milieu. Le polygone est terminé si l'utilisateur
         * clique sur le bouton droit.
         */
        if (currentStep == 0)
        {
            if (e.getButton() == MouseEvent.BUTTON1)
            {
                // On initie le polygone
                startFigure(e);
                System.out.println("initating polygon");
            }
        }
        else
        {
            // Polygon poly = (Polygon) drawingModel.getLastFigure();
            Polygon poly = (Polygon) currentFigure;

            switch (e.getButton())
            {
                case MouseEvent.BUTTON1:
                    // On ajoute un point au polygone
                    poly.addPoint(p.x, p.y);
                    break;
                case MouseEvent.BUTTON2:
                    // On supprime le dernier point
                    poly.removeLastPoint();
                    break;
                case MouseEvent.BUTTON3:
                    // On termine le polygone
                    endFigure(e);
                    break;
            }
        }

        drawingModel.update();
        updateTip();
    }

    /*
     * (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseEntered(MouseEvent e)
    {
        // Rien
    }

    /*
     * (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseExited(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseExited(MouseEvent e)
    {
        // Rien
    }

    /*
     * (non-Javadoc)
     * @see
     * java.awt.event.MouseMotionListener#mouseDragged(java.awt.event.MouseEvent
     * )
     */
    @Override
    public void mouseDragged(MouseEvent e)
    {
        // Rien
    }
```

```
165
166     /**
167      * Déplacement du dernier point du polygon si l'on se trouve à la seconde
168      * étape de la création
169      * @param e L'évènement souris associé
170      * @see
171      * java.awt.event.MouseMotionListener#mouseMoved(java.awt.event.MouseEvent)
172      */
173     @Override
174     public void mouseMoved(MouseEvent e)
175     {
176         /*
177          * déplacement du dernier point du polygone si l'on est toujours en
178          * cours de création du polygone, rien sinon
179          */
180         if (currentStep > 0)
181         {
182             // AbstractFigure figure = drawingModel.getLastFigure();
183             Figure figure = currentFigure;
184             if (figure ≠ null)
185             {
186                 figure.setLastPoint(e.getPoint());
187             }
188             drawingModel.update();
189         }
190     }
191 }
```



```
1  /**
2   * Package contenant les différents widgets (éléments graphiques)
3   */
4  package widgets;
```

```java
package figures.enums;

import java.awt.BasicStroke;
import java.awt.Paint;
import java.awt.Point;
import java.awt.geom.Point2D;

import javax.swing.JLabel;

import figures.Circle;
import figures.Drawing;
import figures.Ellipse;
import figures.Figure;
import figures.Polygon;
import figures.Rectangle;
import figures.RoundedRectangle;
import figures.creationListeners.AbstractCreationListener;
import figures.creationListeners.PolygonCreationListener;
import figures.creationListeners.RectangularShapeCreationListener;
import figures.creationListeners.RoundedRectangleCreationListener;

/**
 * Enumeration des différentes figures possibles
 * @author davidroussel
 */
public enum FigureType
{
    /**
     * Les différents types de figures
     */
    CIRCLE,
    ELLIPSE,
    RECTANGLE,
    ROUNDED_RECTANGLE,
    POLYGON,
    NONE;

    /**
     * Nombre de figures référencées ici (à changer si on ajoute des types de
     * figures)
     */
    public final static int NbFigureTypes = 5;

    /**
     * Obtention d'une instance de figure correspondant au type
     *
     * @param stroke la césure du trait (ou pas de trait si null)
     * @param edge la couleur du trait (ou pas de trait si null)
     * @param fill la couleur de remplissage (ou pas de remplissage si null)
     * @param x l'abcisse du premier point de la figure
     * @param y l'ordonnée du premier point de la figure
     * @return une nouvelle instance correspondant à la valeur de cet enum
     * @throws AssertionError si la valeur de cet enum n'est pas prévue
     */
    public Figure getFigure(BasicStroke stroke,
            Paint edge,
            Paint fill,
            Point2D p) throws AssertionError
    {
        switch (this)
        {
            case CIRCLE:
                return new Circle(stroke, edge, fill, p, 0.0f);
            case ELLIPSE:
                return new Ellipse(stroke, edge, fill, p, p);
            case RECTANGLE:
                return new Rectangle(stroke, edge, fill, p, p);
            case ROUNDED_RECTANGLE:
                return new RoundedRectangle(stroke, edge, fill, p, p, 0);
            case POLYGON:
                Point pp = new Point(Double.valueOf(p.getX()).intValue(),
                        Double.valueOf(p.getY()).intValue());
                return new Polygon(stroke, edge, fill, pp, pp);
            case NONE:
                return null;
        }

        throw new AssertionError("FigureType unknown assertion: " + this);
    }

    /**
     * Obtention d'un CreationListener adequat pour la valeur de cet enum
```



```java
     *
     * @param model le modèle de dessin à modifier
     * @param tipLabel le label dans lequel afficher les conseils utilisateur
     * @return une nouvelle instance de CreationListener adéquate pour le type
     *         de figure de cet enum.
     * @throws AssertionError si la valeur de cet enum n'est pas prévue
     */
    public AbstractCreationListener getCreationListener(Drawing model,
            JLabel tipLabel) throws AssertionError
    {
        switch (this)
        {
            case CIRCLE:
            case ELLIPSE:
            case RECTANGLE:
                return new RectangularShapeCreationListener(model, tipLabel);
            case ROUNDED_RECTANGLE:
                return new RoundedRectangleCreationListener(model, tipLabel);
            case POLYGON:
                return new PolygonCreationListener(model, tipLabel);
            case NONE:
                return null;
        }

        throw new AssertionError("FigureType unknown assertion: " + this);
    }

    /**
     * Représentation sous forme de chaine de caractères
     *
     * @return une chaine de caractère représentant la valeur de cet enum
     * @throws AssertionError si la valeur de cet enum n'est pas prévue
     */
    @Override
    public String toString() throws AssertionError
    {
        switch (this)
        {
            case CIRCLE:
                return new String("Circle");
            case ELLIPSE:
                return new String("Ellipse");
            case RECTANGLE:
                return new String("Rectangle");
            case ROUNDED_RECTANGLE:
                return new String("Rounded Rectangle");
            case POLYGON:
                return new String("Polygon");
            case NONE:
                return new String("None");
        }

        throw new AssertionError("FigureType unknown assertion: " + this);
    }

    /**
     * Otention d'un tableau de chaine de caractères contenant l'ensemble des
     * nom des figures
     *
     * @return un tableau de chaine de caractères contenant l'ensemble des nom
     *         des figures
     */
    public static String[] stringValues()
    {
        FigureType[] values = FigureType.values();
        String[] stringValues = new String[values.length - 1]; // Except NONE

        for (int i = 0; i < stringValues.length; i++)
        {
            stringValues[i] = values[i].toString();
        }

        return stringValues;
    }

    /**
     * Conversion d'un entier en FigureType
     *
     * @param i l'entier à convertir en FigureType
     * @return le FigureType correspondant à l'entier
     */
    public static FigureType fromInteger(int i)
```

```
165     {
166         switch (i)
167         {
168             case 0:
169                 return CIRCLE;
170             case 1:
171                 return ELLIPSE;
172             case 2:
173                 return RECTANGLE;
174             case 3:
175                 return ROUNDED_RECTANGLE;
176             case 4:
177                 return POLYGON;
178             default:
179                 return POLYGON;
180         }
181     }
182 }
```



```
1  package figures.enums;
2
3  import java.awt.BasicStroke;
4
5  /**
6   * Le type de trait des lignes (continu, pointillé, ou sans trait)
7   * @author davidroussel
8   */
9  public enum LineType
10 {
11     /**
12      * Pas de trait
13      */
14     NONE,
15     /**
16      * Trait plein
17      */
18     SOLID,
19     /**
20      * Trait pointillé
21      */
22     DASHED;
23
24     /**
25      * Le nombre de type de lignes (à changer si l'on rajoute un type de ligne)
26      */
27     public static final int NbLineTypes = 3;
28
29     /**
30      * Conversion d'un entier vers un {@link LineType}.
31      * A utiliser pour convertir l'index de l'élément sélectionné d'un combobox
32      * dans le type de ligne correspondant
33      * @param i l'entier à convertir
34      * @return le LineType correspondant
35      */
36     public static LineType fromInteger(int i)
37     {
38         switch (i)
39         {
40             case 0:
41                 return NONE;
42             case 1:
43                 return SOLID;
44             case 2:
45                 return DASHED;
46             default:
47                 return NONE;
48         }
49     }
50
51     /**
52      * Conversion d'un {@link BasicStroke} en type de ligne
53      * @param stroke le stroke à examiner
54      * @return le type de ligne correspondant (NONE si le stroke est nul,
55      * SOLID si le stroke ne contient pas de dash array, DASHED si le stroke
56      * contient un dash array.
57      */
58     public static LineType fromStroke(BasicStroke stroke)
59     {
60         if (stroke == null)
61         {
62             return LineType.NONE;
63         }
64         else
65         {
66             float[] dashArray = stroke.getDashArray();
67             if (dashArray == null)
68             {
69                 return LineType.SOLID;
70             }
71             else
72             {
73                 return LineType.DASHED;
74             }
75         }
76     }
77
78     /**
79      * Représentation sous forme de chaine de caractères
80      * @return une chaine de caractères représentant la valeur de cet enum
81      */
82     @Override
```

```
83      public String toString() throws AssertionError
84      {
85          switch (this)
86          {
87              case NONE:
88                  return new String("None");
89              case SOLID:
90                  return new String("Solid");
91              case DASHED:
92                  return new String("Dashed");
93          }
94
95          throw new AssertionError("LineType Unknown assertion " + this);
96      }
97
98      /**
99       * Obtention d'un tableau de string contenant tous les noms des types.
100      * A utiliser lors de la création d'un combobox avec :
101      * LineType.stringValues()
102      * @return un tableau de string contenant tous les noms des types
103      */
104     public static String[] stringValues()
105     {
106         LineType[] values = LineType.values();
107         String[] stringValues = new String[values.length];
108         for (int i = 0; i < values.length; i++)
109         {
110             stringValues[i] = values[i].toString();
111         }
112
113         return stringValues;
114     }
115 }
```



```
1   package figures.enums;
2
3   import java.awt.Paint;
4
5   import figures.Drawing;
6
7   /**
8    * Enumeration de ce à quoi s'applique une couleur ({@link Paint}) à utiliser
9    * dans le {@link widgets.EditorFrame.ColoItemListener}
10   *
11   * @author davidroussel
12   */
13  public enum PaintToType
14  {
15      /**
16       * La couleur s'applique au remplissage
17       */
18      FILL,
19      /**
20       * La couleur s'applique au trait
21       */
22      EDGE;
23
24      /**
25       * Application d'une couleur au modèle de dessin en fonction de la valeur de
26       * l'enum
27       *
28       * @param paint la couleur à appliquer
29       * @param drawing le modèle de dessin sur lequel appliquer la couleur
30       * @throws AssertionError si le type de l'enum est inconnu
31       */
32      public void applyPaintTo(Paint paint, Drawing drawing)
33              throws AssertionError
34      {
35          switch (this)
36          {
37              case FILL:
38                  drawing.setFillPaint(paint);
39                  break;
40              case EDGE:
41                  drawing.setEdgePaint(paint);
42                  break;
43              default:
44                  throw new AssertionError(
45                          "PaintApplicationType unknown assertion " + this);
46          }
47      }
48
49      /**
50       * Représentation sous forme de chaine de caractères
51       *
52       * @return une chaine de caractères représentant la valeur de cet enum
53       */
54      @Override
55      public String toString() throws AssertionError
56      {
57          switch (this)
58          {
59              case FILL:
60                  return new String("Fill");
61              case EDGE:
62                  return new String("Edge");
63          }
64
65          throw new AssertionError("PaintApplicationType Unknown assertion "
66                  + this);
67      }
68
69  }
```

```java
/**
 * Package contenant les différents widgets (éléments graphiques)
 */
package widgets;
```

29 nov. 15 17:48 **package−info.java** Page 1/1


```java
/**
 * Package contenant les différents widgets (éléments graphiques)
 */
package widgets;
```

```java
package filters;

import java.util.function.Predicate;

import figures.Figure;

/**
 * Prédicat permettant de filtrer les figures à partir d'un élément de type T.
 * T pourra être instancié avec divers types dans les classes filles pour
 * filtrer :
 * <ul>
 *  <li>le type de figures: {@link figures.enums.FigureType}</li>
 *  <li>la couleur de remplissage ou de trait: {@link java.awt.Paint}</li>
 *  <li>le type de trait: {@link figures.enums.LineType}</li>
 * </ul>
 * @author davidroussel
 */
public class FigureFilter<T> implements Predicate<Figure>
{
    /**
     * L'élément sur lequel filter les figures
     */
    protected T element;

    /**
     * Constructeur par défaut
     */
    public FigureFilter()
    {
        element = null;
    }

    /**
     * Constructeur d'un figure filter
     * @param element l'élément de référence du prédicat
     */
    public FigureFilter(T element)
    {
        this.element = element;
    }

    /**
     * Accesseur à l'élément du filtre
     * @return l'élément du filtre
     */
    public T getElement()
    {
        return element;
    }

    /**
     * Test du prédicat
     * @param f la figure à tester
     * @return vrai si un élément de la figure correspond à l'élément contenu
     * dans ce prédicat
     * @note Cette méthode DOIT être réimplémentée dans les classes filles pour
     * renvoyer autre chose que false
     * @see java.util.function.Predicate#test(java.lang.Object)
     */
    @Override
    public boolean test(Figure f)
    {
        return false;
    }

    /**
     * Comparaison avec un autr objet
     * @param obj l'objet à comparer
     * @return true si l'autre objet est un filtre sur le même type d'élément
     */
    @Override
    public boolean equals(Object obj)
    {
        if (obj ≡ null)
        {
            return  false;
        }

        if (obj ≡ this)
        {
            return true;
        }
```



```java

        if (obj instanceof FigureFilter<?>)
        {
            FigureFilter<?> ff = (FigureFilter<?>) obj;
            if (ff.element ≠ null ∧ element ≠ null)
            {
                if (ff.element.getClass() ≡ element.getClass())
                {
                    @SuppressWarnings("unchecked")
                    FigureFilter<T> fft = (FigureFilter<T>)ff;
                    return element.equals(fft.element);
                }
                else
                {
                    if (element ≠ null ∨ ff.element ≠ null)
                    {
                        return false;
                    }
                    else
                    {
                        return true;
                    }
                }
            }
        }

        return false;
    }

    /**
     * Chaîne de caractères représentant le filtre
     * @return une chaine de caractère représentantn le filtre
     */
    @Override
    public String toString()
    {
        return new String(getClass().getSimpleName() + "<"
            + (element ≠ null ? element.getClass().getSimpleName() : "null")
            + ">(" + (element ≠ null ? element.toString() : "") + ")");
    }
}
```

```java
package filters;

import java.util.Collection;
import java.util.Iterator;
import java.util.Vector;
import java.util.function.Predicate;

import figures.Figure;

/**
 * Collection de filtres
 * @author davidroussel
 */
public class FigureFilters<T> implements Collection<FigureFilter<T>>, Predicate<Figure>
{
    /**
     * Vecteur de filtres
     */
    private Vector<FigureFilter<T>> filters;

    /**
     * Constructeur par défaut
     */
    public FigureFilters()
    {
        filters = new Vector<FigureFilter<T>>();
    }

    /**
     * Test du prédicat
     * @param f la figure à tester
     * @return true si l'un au moins des prédicats de la collection est vrai,
     * false sinon
     * @see filters.FigureFilter#test(figures.Figure)
     */
    @Override
    public boolean test(Figure f)
    {
        boolean result = false;

        for (FigureFilter<T> ff : this)
        {
            boolean thisResult = ff.test(f);
            // System.out.println(ff + (thisResult ? "passed": "denied"));
            result |= thisResult;
        }

        // System.out.println(this +  (result ? "passed": "denied"));

        return result;
    }

    /**
     * Taille de la collection
     * @return la taille de la collection
     */
    @Override
    public int size()
    {
        return filters.size();
    }

    /**
     * Teste si la collection est vide
     * @return true si la collection est vide
     */
    @Override
    public boolean isEmpty()
    {
        return filters.isEmpty();
    }

    /**
     * Test de contenu d'un objet dans la collection de filtres
     * @param o l'objet recherché dans la collection de filtres
     * @return true si l'objet est contenu dans la collection de filtres
     */
    @Override
    public boolean contains(Object o)
    {
        return filters.contains(o);
    }
```



```java

    /**
     * Itérateur de la collection de {@link FigureFilter}
     * @return l'itérateur sur les filtres de la collection
     */
    @Override
    public Iterator<FigureFilter<T>> iterator()
    {
        return filters.iterator();
    }

    /**
     * Conversion en tableau d'objets
     * @return un tableau d'objets contenant les éléments de la collection
     */
    @Override
    public Object[] toArray()
    {
        return filters.toArray();
    }

    /**
     * Conversion en tableau générique
     * @param a un tableau générique spécimen
     * @return un tableau générique contenant les éléments de la collection
     */
    @Override
    @SuppressWarnings("hiding")
    public <T> T[] toArray(T[] a)
    {
        return filters.toArray(a);
    }

    /**
     * Ajout d'un nouveau filtre à la collection uniquement si celle ci ne
     * contient pas déjà ce filtre
     * @param filter le filtre à ajouter
     * @return true si le filtre à ajouté n'était pas déjà présent dans la
     * collection et qu'il a été ajouté
     */
    @Override
    public boolean add(FigureFilter<T> filter)
    {
        if (¬contains(filter))
        {
            return filters.add(filter);
        }
        else
        {
            return false;
        }
    }

    /**
     * Retrait d'un objet de la collection
     * @param o l'objet à retirer de la collection
     * @return true si l'objet a été retiré de la collection
     */
    @Override
    public boolean remove(Object o)
    {
        return filters.remove(o);
    }

    /**
     * Test si une collection est entièrement contenue dans la collection
     * @param c la collection à tester
     * @return true si la collection c est entièrement contenue dans la
     * collection
     */
    @Override
    public boolean containsAll(Collection<?> c)
    {
        return filters.containsAll(c);
    }

    /**
     * Ajout d'une collection de {@link FigureFilters} à la collection courante
     * @param c la collection de {@link FigureFilter} à ajouter
     * @return true si au moins un élément de la collection c a été ajouté
     * à la collection courante
     */
```

```
165     @Override
166     public boolean addAll(Collection<? extends FigureFilter<T>> c)
167     {
168         boolean added = false;
169         for (FigureFilter<T> ff : c)
170         {
171             if (¬contains(ff))
172             {
173                 added |= add(ff);
174             }
175         }
176 
177         return added;
178     }
179 
180     /**
181      * Retrait de tous les éléments d'une collection de la collection courante
182      * @param c la collection à retirer de la collection courante
183      * @return true si la collection courante a été modifiée par cette opération
184      */
185     @Override
186     public boolean removeAll(Collection<?> c)
187     {
188         return filters.removeAll(c);
189     }
190 
191     /**
192      * Conservation dans la collection courante uniquement des éléments présents
193      * dans la collection c
194      * @param c la collection qui détermine les éléments à conserver dans la
195      * collection courante
196      * @return true si la collection courante a été modifiée par cette opération
197      */
198     @Override
199     public boolean retainAll(Collection<?> c)
200     {
201         boolean retained = filters.retainAll(c);
202 
203         // remove doubles
204 
205         return retained;
206     }
207 
208     /**
209      * Effacement de la collection
210      */
211     @Override
212     public void clear()
213     {
214         filters.clear();
215     }
216 
217     /**
218      * Représentation de la collection de filtres
219      * @return une chaine de caractères représentantn tous les filtres
220      */
221     @Override
222     public String toString()
223     {
224         StringBuilder sb = new StringBuilder();
225 
226         sb.append(getClass().getSimpleName());
227         sb.append("[");
228         sb.append(filters.size());
229         sb.append("]\n");
230         for ( FigureFilter<T> ff : filters)
231         {
232             sb.append(ff.toString() + "\n");
233         }
234 
235         return sb.toString();
236     }
237 
238 
239 }
```



```
1   package filters;
2 
3   import java.awt.Paint;
4 
5   import figures.Figure;
6 
7   /**
8    * Filtre filtrant les figures possédant une certaine couleur de trait
9    * @author davidroussel
10   */
11  public class EdgeColorFilter extends FigureFilter<Paint>
12  {
13      /**
14       * Constructeur d'un {@link EdgeColorFilter}
15       * @param paint la couleur à filtrer
16       */
17      public EdgeColorFilter(Paint paint)
18      {
19          super(paint);
20      }
21 
22      /**
23       * Test du prédicat
24       * @return true si la figure courante possède la même couleur de trait
25       */
26      @Override
27      public boolean test(Figure f)
28      {
29          return f.getEdgePaint().equals(element);
30      }
31 
32  }
```

```java
package filters;

import java.awt.Paint;

import figures.Figure;

/**
 * Filtre filtrant les figures possédant une certaine couleur de remplissage
 * @author davidroussel
 */
public class FillColorFilter extends FigureFilter<Paint>
{
    /**
     * Constructeur d'un {@link FillColorFilter}
     * @param paint la couleur à filtrer
     */
    public FillColorFilter(Paint paint)
    {
        super(paint);
    }

    /**
     * Test du prédicat
     * @return true si la figure courante possède la même couleur de remplissage
     */
    @Override
    public boolean test(Figure f)
    {
        return f.getFillPaint().equals(element);
    }

}
```



```java
package filters;

import figures.Figure;
import figures.enums.LineType;

/**
 * Filtre filtrant les figures ayant un certain type de trait
 * @author davidroussel
 */
public class LineFilter extends FigureFilter<LineType>
{
    /**
     * Constructeur d'un {@link LineFilter}
     * @param type le type de ligne à filtrer
     */
    public LineFilter(LineType type)
    {
        super(type);
    }

    /**
     * Test du prédicat
     * @return true si la figure courante possède le même type de trait
     */
    @Override
    public boolean test(Figure f)
    {
        return LineType.fromStroke(f.getStroke()) == element;
    }
}
```

```java
package filters;

import figures.Figure;
import figures.enums.FigureType;

/**
 * Filtre de figure appliqué au type de figure
 * @author davidroussel
 */
public class ShapeFilter extends FigureFilter<FigureType>
{
    /**
     * Constructeur d'un {@link ShapeFilter} à partir d'un type de figure
     * @param element le type de figure à tester par ce prédicat
     */
    public ShapeFilter(FigureType element)
    {
        super(element);
    }

    /**
     * Test du prédicat
     * @param f la figure à tester
     * @return vrai si la figure f est du type contenu dans element
     * @see java.util.function.Predicate#test(java.lang.Object)
     */
    @Override
    public boolean test(Figure f)
    {
        return f.getType() ≡ element;
    }
}
```



```java
package utils;

import java.util.HashMap;

/**
 * Flyweight gérant les différents éléments utilisés dans la zone de dessin.
 * Utilisable avec les {@link Paint} et avec les {@link BasicStroke} des figures
 * Gère les éléments dans une HashMap<Integer, T> dont la clé correspond au
 * hashCode de l'élément correspondant. Lorsque l'on demande un élément à la
 * Factory, celui ci le recherche dans sa table de hachage : Si l'élément n'est
 * pas déjà présent dans la table de hachage il est ajouté, puis renvoyé, s'il
 * est déjà présent dans la table de hachage il est directement renvoyé et celui
 * demandé est alors destructible par le garbage collector.
 *
 * @author davidroussel
 */
public class FlyweightFactory<T>
{
    /**
     * La table de hachage contenant les différentes paires <hashcode,elt> et
     * dont les clés sont les hashCode des différents éléments.
     */
    protected HashMap<Integer, T> map;

    /**
     * Constructeur d'un FlyweightFactory.
     * Initialise la {@link HashMap}
     */
    public FlyweightFactory()
    {
        map = new HashMap<Integer, T>();
    }

    /**
     * Obtention d'un élément à partir son hashcode plutôt que par l'élément
     * lui même
     * @param hash le hachage de l'élément demandé
     * @return l'élément correspondant au hachage demandé ou bien null si aucun
     * élément avec ce hachage n'est contenu dans la factory
     * @note cette méthode est nécessaire lorsque l'on veut stocker dans la
     * factory des éléments qui ne réimplémentent pas la méthode hashCode.
     * Auquel cas on fournit soi même un code de hachage.
     */
    protected T get(int hash)
    {
        Integer key = Integer.valueOf(hash);
        if (map.containsKey(key))
        {
            return map.get(key);
        }

        return null;
    }

    /**
     * Ajout d'un élément à la factory en fournissant un hashcode particulier
     * @param hash le hachage voulu pour cet élément
     * @param element l'élément à ajouter
     * @return true si aucun élément avec ce hachage n'était contenu dans la
     * factory et que le couple hash/value a bien été ajouté à la factory
     * @note cette méthode est nécessaire lorsque l'on veut stocker dans la
     * factory des éléments qui ne réimplémentent pas la méthode hashCode.
     * Auquel cas on fournit soi même un code de hachage.
     */
    protected boolean put(int hash, T element)
    {
        Integer key = Integer.valueOf(hash);
        if (¬map.containsKey(key))
        {
            if (element ≠ null)
            {
                map.put(key, element);
                System.out.println("Added " + element
                    + " to the flyweight factory which contains "
                    + map.size() + " elements");
                return true;
            }
            else
            {
                System.err.println("FlyweightFactory::put(...) : null element");
            }
        }
```

```
83          return false;
84      }
85
86      /**
87       * Obtention d'un élément (nouveau ou pas) : Lorsque l'élément demandé est
88       * déjà présent dans la table on le renvoie directement sinon celui ci est
89       * ajouté à la table avant d'être renvoyé
90       * @param element l'élément demandé [celui ci pourra être détruit par le
91       * garbage collector si il en existe déjà un équivalent dans la table]
92       * @return l'élément demandé en provenance de la table
93       */
94      public T get(T element)
95      {
96          if (element ≠ null)
97          {
98              int hash = element.hashCode();
99              T result = get(hash);
100             if (result ≡ null)
101             {
102                 put(hash, element);
103                 result = get(hash);
104             }
105             return result;
106         }
107         return null;
108     }
109
110     /**
111      * Nettoyage de tous les éléments
112      */
113     public void clear()
114     {
115         map.clear();
116     }
117
118     /**
119      * Nettoyage avant destruction de la factory
120      */
121     @Override
122     protected void finalize()
123     {
124         clear();
125     }
126 }
```



```
1   package utils;
2
3   import javax.swing.ImageIcon;
4
5   /**
6    * Classe contenant une FlyweightFactory pour les les icônes, afin de pouvoir
7    * réutiliser une même icône (chargée à partir d'un fichier image contenu dans
8    * le package "images") à plusieurs endroits de l'interface graphique.
9    * @author davidroussel
10   */
11  public class IconFactory
12  {
13      /**
14       * le répertoire de base pour chercher les images
15       */
16      private final static String ImageBase = "/images/";
17
18      /**
19       * L'extension par défaut pour chercher les fichiers images
20       */
21      private final static String ImageType = ".png";
22
23      /**
24       * La factory stockant et fournissant les icônes
25       */
26      static private FlyweightFactory<ImageIcon> iconFactory =
27          new FlyweightFactory<ImageIcon>();
28
29      /**
30       * Méthode d'obtention d'une icône pour un nom donné
31       * @param name le nom de l'icône que l'on recherche
32       * @return l'icône correspondant au nom demandé si un fichier avec ce nom
33       * est trouvé dans le package/répertoire "images" ou bien null si aucune
34       * image correspondant à ce nom n'est trouvée.
35       */
36      static public ImageIcon getIcon(String name)
37      {
38          // checks if there is an icon with this name in the "images" directory
39          if (name.length() > 0)
40          {
41              int hash = name.hashCode();
42              ImageIcon icon = iconFactory.get(hash);
43              if (icon ≡ null)
44              {
45                  icon = new ImageIcon(IconFactory.class
46                      .getResource(ImageBase + name + ImageType));
47                  if (icon.getImageLoadStatus() ≡ java.awt.MediaTracker.COMPLETE)
48                  {
49                      icon.setDescription(name);
50                      iconFactory.put(hash, icon);
51                  }
52                  else
53                  {
54                      System.err.println("IconFactory::getIcon(" + name
55                          + "): could'nt find file " + ImageBase + name
56                          + ImageType);
57                  }
58
59                  return iconFactory.get(hash);
60              }
61              else
62              {
63                  return icon;
64              }
65          }
66          else
67          {
68              System.err.println("IconFactory::getIcon(<EMPTY NAME>)");
69          }
70
71          return null;
72      }
73  }
```

```java
package utils;

import javax.swing.ImageIcon;

/**
 * Class defining an item Name associated to an Icon
 * @author davidroussel
 */
public class IconItem
{
    /**
     * Combobox item name
     */
    private String caption;

    /**
     * Combobox item icon
     * @note typically reflects the item name in a file named <caption>.png
     */
    private ImageIcon icon;

    /**
     * Constructor from caption only
     * @param caption the caption of this item
     */
    public IconItem(String caption)
    {
        this.caption = caption;
        icon = IconFactory.getIcon(caption);
        if (icon == null)
        {
            System.err.println("IconItem(" + caption
                + "): could not find corresponding icon");
        }
    }

    /**
     * Caption accessor
     * @return the caption of this item
     */
    public String getCaption()
    {
        return caption;
    }

    /**
     * Icon accessor
     * @return the icon of this item
     */
    public ImageIcon getIcon()
    {
        return icon;
    }
}
```



```java
package utils;

import java.awt.Color;
import java.awt.Component;
import java.awt.Paint;
import java.util.HashMap;
import java.util.Map;

import javax.swing.JColorChooser;

/**
 * Classe contenant une FlyweightFactory pour les {@link Paint} afin de pouvoir
 * réutiliser un même {@link Paint} à plusieurs endroits du programme
 * @author davidroussel
 */
public class PaintFactory
{
    /**
     * Map associant des noms de couleurs standard à des {@link Paint} standards
     */
    private static final Map<String, Paint> standardPaints = fillStandardPaints();

    /**
     * Construction de la map des {@link Paint} standards
     * @return une map contenant les {@link Paint} standards
     */
    private static Map<String, Paint> fillStandardPaints()
    {
        Map<String, Paint> map = new HashMap<String, Paint>();
        map.put("Black", Color.black);
        map.put("Blue", Color.blue);
        map.put("Cyan", Color.cyan);
        map.put("Green", Color.green);
        map.put("Magenta", Color.magenta);
        map.put("None", null);
        map.put("Orange", Color.orange);
        map.put("Pink", Color.pink);
        map.put("Red", Color.red);
        map.put("White", Color.white);
        map.put("Yellow", Color.yellow);

        return map;
    }

    /**
     * Flyweight factory stockant tous les {@link Paint} déjà requis
     */
    private static FlyweightFactory<Paint> paintFactory =
        new FlyweightFactory<Paint>();

    /**
     * Obtention d'un {@link Paint} de la factory
     * @param paint le paint recherché
     * @return le paint recherché extrait de la factory
     */
    public static Paint getPaint(Paint paint)
    {
        if (paint != null)
        {
            return paintFactory.get(paint);
        }

        return null;
    }

    /**
     * Obtention d'un paint de la factory par son nom en le recherchant dans les
     * {@link #standardPaints}
     * @param paintName le nom de la couleur requise
     * @return le paint recherché extrait de la factory
     */
    public static Paint getPaint(String paintName)
    {
        if (paintName.length() > 0)
        {
            if (standardPaints.containsKey(paintName))
            {
                return paintFactory.get(standardPaints.get(paintName));
            }
        }

        return null;
```

```
83      }
84
85      /**
86       * Obtention d'un paint de la factory en déclenchant une boite de dialogue
87       * de choix d'une couleur.
88       * @param component le composant AWT à l'origine de la boite de dialogue
89       * @param title le titre de la boite de dialogue
90       * @param initialColor la couleur initiale de la boite de dialogue de choix
91       * de couleus
92       * @return
93       */
94      public static Paint getPaint(Component component,
95                                   String title,
96                                   Color initialColor)
97      {
98          if (component ≠ null)
99          {
100             Color color = JColorChooser.showDialog(component, title, initialColor);
101             if (color ≠ null)
102             {
103                 return paintFactory.get(color);
104             }
105         }
106
107         return null;
108     }
109 }
```



```
1   package utils;
2
3   import java.awt.BasicStroke;
4
5   import figures.enums.LineType;
6
7   /**
8    * Classe contenant une FlyweightFactory pour les {@link BasicStroke} afin de pouvoir
9    * réutiliser un même {@link BasicStroke} à plusieurs endroits du programme
10   * @author davidroussel
11   */
12  public class StrokeFactory
13  {
14      /**
15       * Flyweight factory stockant tous les {@link BasicStroke} déjà requis
16       */
17      private static FlyweightFactory<BasicStroke> strokeFactory =
18          new FlyweightFactory<BasicStroke>();
19
20      /**
21       * Obtention d'un {@link BasicStroke} de la factory
22       * @param stroke le paint recherché
23       * @return le stroke recherché
24       */
25      public static BasicStroke getStroke(BasicStroke stroke)
26      {
27          if (stroke ≠ null)
28          {
29              return strokeFactory.get(stroke);
30          }
31
32          return null;
33      }
34      /**
35       * Obtention d'un {@link BasicStroke} à partir d'un type de trait et
36       * d'une épaisseur de trait
37       * @param type le type de trait (NONE, SOLID ou DASHED)
38       * @param width l'épaisseur du trait
39       * @return une {@link BasicStroke} correspondant au type et à l'épaisseur
40       * de trait en provenance de la factory
41       */
42      public static BasicStroke getStroke(LineType type, float width)
43      {
44          switch (type)
45          {
46              default:
47              case NONE:
48                  return null;
49              case SOLID:
50                  return getStroke(new BasicStroke(width,
51                                                   BasicStroke.CAP_ROUND,
52                                                   BasicStroke.JOIN_ROUND));
53              case DASHED:
54                  final float dash1[] = { 2 * width };
55                  return getStroke(new BasicStroke(width,
56                                                   BasicStroke.CAP_ROUND,
57                                                   BasicStroke.JOIN_ROUND,
58                                                   width, dash1, 0.0f));
59          }
60
61      }
62  }
```

```java
/**
 * Package contenant les différents widgets (éléments graphiques)
 */
package widgets;
```



```java
package widgets;

import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.RenderingHints;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.geom.Point2D;
import java.text.DecimalFormat;
import java.util.Observable;
import java.util.Observer;

import javax.swing.JLabel;
import javax.swing.JPanel;

import figures.Drawing;
import figures.Figure;
import figures.creationListeners.AbstractCreationListener;

/**
 * Panel de dessin des figures (Vue): mis à jour par modèle des figures (
 * {@link Drawing}) au travers d'un observateur. On attache des Listeners
 * (Controleurs) à ce Panel pour :
 * <dl>
 * <dt>Attachements statiques :</dt>
 * <dd>Mettre à jour les coordonnées du pointeur de la souris dans la barre
 * d'état : {@link #coordLabel}</dd>
 * <dd>Mettre à jour le panneau d'informations relatif aux figures située sous
 * le pointeur de la souris : {@link #infoPanel}.</dd>
 * <dt>Attachements dynamique :</dt>
 * <dd>Pour chaque type de figure à créer on attache un
 * {@link AbstractCreationListener} ou plus exactement un de ses descendants
 * pour traduire les évènements souris en instructions pour le modèle de dessin
 * lors de la création d'une nouvelle figure.
 * </dl>
 *
 * @author davidroussel
 */
public class DrawingPanel extends JPanel implements Observer, MouseListener,
        MouseMotionListener
{
    /**
     * Taille effective du panel. Ce panel n'ayant pas de Layout Manager, il est
     * important de conserver une taille effective qui puisse être renvoyée dans
     * la méthode {@link #getPreferredSize()} et modifiée par un
     * {@link java.awt.event.ComponentListener} tel que le
     * {@link ResizeListener} ci-dessous.
     */
    protected Dimension size;

    /**
     * Contrôleur de changement de taille afin de mettre à jour
     * {@link DrawingPanel#size} utilisé dans
     * {@link DrawingPanel#getPreferredSize()}.
     *
     * @author davidroussel
     */
    protected class ResizeListener extends ComponentAdapter
    {
        /**
         * Action à réaliser lorsque le composant change de taille
         */
        @Override
        public void componentResized(ComponentEvent e)
        {
            size = e.getComponent().getSize();
        }
    }

    /**
     * Le modèle (les figures) à dessiner
     */
    private Drawing drawingModel;

    /**
```

```
83      * Le label (qq part dans la GUI) dans lequel afficher les coordonnées du
84      * pointeur de la souris
85      */
86     private JLabel coordLabel;
87
88     /**
89      * L'{@link InfoPanel} dans lequel afficher les informations à propos de
90      * la figure sous le curseur.
91      */
92     private InfoPanel infoPanel;
93
94     /**
95      * Chaîne de caractère à afficher par défaut dans le {@link #coordLabel}
96      */
97     public final static String defaultCoordString = new String("x: ___ y: ___");
98
99     /**
100     * Le formatteur à utiliser pour formater les nombres dans le
101     * {@link #coordLabel} et dans l'{@link #infoPanel}
102     */
103    private final static DecimalFormat coordFormat = new DecimalFormat("000");
104
105    /**
106     * état indiquant s'il faut envoyer les coordonnées de la souris ou la
107     * figure au dessus de laquelle se trouve la souris. Lorsque le curseur sort
108     * du widget (mouseExited) on cesse d'envoyer les coordonnées de la souris
109     * et lorsqu'elle entre (mouseEntered) on recommence à envoyer les
110     * coordonnées de la souris.
111     */
112    private boolean sendInfoState;
113
114    /**
115     * Constructeur de la zone de dessin à partir d'un modèle de dessin.
116     *
117     * @param drawing le modèle de dessin
118     * @param coordLabel le label à mettre à jour avec les coordonnées du
119     *            curseur de la souris
120     * @param infoPanel le panneau d'information des figures à mettre à jour
121     *            avec les informations relative à la figure située sous le
122     *            curseur de la souris
123     */
124    public DrawingPanel(Drawing drawing, JLabel coordLabel, InfoPanel infoPanel)
125    {
126        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
127        size = new Dimension(800, 600);
128        setPreferredSize(size);
129        addComponentListener(new ResizeListener());
130
131        setBackground(Color.WHITE);
132        setLayout(null);
133        setDoubleBuffered(true);
134
135        drawingModel = drawing;
136        if (drawing ≠ null)
137        {
138            drawingModel.addObserver(this);
139        }
140        else
141        {
142            System.err.println("DrawingPanel caution: null drawing");
143        }
144
145        this.coordLabel = coordLabel;
146
147        if (this.coordLabel ≠ null)
148        {
149            this.coordLabel.setText(defaultCoordString);
150        }
151        else
152        {
153            System.err.println("DrawingPanel : null coordLabel");
154        }
155
156        this.infoPanel = infoPanel;
157
158        if (this.infoPanel ≠ null)
159        {
160            this.infoPanel.resetLabels();
161        }
162        else
163        {
164            System.err.println("DrawingPanel : null infoPanel");
```



```
165        }
166
167        // DrawingPanel est son propre listener d'évènements souris
168        addMouseListener(this);
169        addMouseMotionListener(this);
170    }
171
172    /**
173     * Accès à la taille effective du panel qui peut changer si celui-ci est
174     * agrandi (avec la fenêtre dans lequel il est par exemple). Cette méthode
175     * permet d'ajuster les scrollbars d'un container qui contiendrait ce panel
176     * lorsque la taille de celui-ci change.
177     *
178     * @return la taille effective du panel de dessin
179     * @see javax.swing.JComponent#getPreferredSize()
180     */
181    @Override
182    public Dimension getPreferredSize()
183    {
184        return size;
185    }
186
187    /**
188     * Mise en place du modèle de dessin. Met en place un nouveau modèle et s'il
189     * est non null ajoute ce panel comme observateur du modèle
190     *
191     * @param drawing le modèle de dessin à mettre en place
192     */
193    public void setDrawing(Drawing drawing)
194    {
195        // retrait du précédent modèle de dessin (s'il existe)
196        if (drawingModel ≠ null)
197        {
198            drawingModel.deleteObserver(this);
199        }
200
201        // Mise en place du nouveau modèle de dessin
202        drawingModel = drawing;
203        if (drawingModel ≠ null)
204        {
205            drawingModel.addObserver(this);
206        }
207    }
208
209    /**
210     * Mise en place du label dans lequel afficher les coordonnées du pointeur
211     * de la souris.
212     *
213     * @param coordLabel le label dans lequel afficher les coordonnées du
214     *            pointeur de la souris.
215     */
216    public void setCoordLabel(JLabel coordLabel)
217    {
218        this.coordLabel = coordLabel;
219    }
220
221    /**
222     * Mise en place du panel d'information dans lequel afficher les infos sur
223     * la figure située sous le curseur
224     *
225     * @param infoPanel l'{@link InfoPanel} à mettre en place
226     */
227    public void setInfoPanel(InfoPanel infoPanel)
228    {
229        this.infoPanel = infoPanel;
230    }
231
232    /**
233     * Dessin du panel. Effacement ce celui-ci puis dessin des figures.
234     * @param g le contexte graphique
235     * @see javax.swing.JComponent#paintComponent(java.awt.Graphics)
236     */
237    @Override
238    protected void paintComponent(Graphics g)
239    {
240        super.paintComponent(g); // Inutile
241
242        // caractéristiques graphiques : mise en place de l'antialiasing
243        Graphics2D g2D = (Graphics2D) g;
244        g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
245                             RenderingHints.VALUE_ANTIALIAS_ON);
246
```

```
247         // taille de la zone de dessin
248         Dimension d = getSize();
249         // on commence par effacer le fond
250         g2D.setColor(getBackground());
251         g2D.fillRect(0, 0, d.width, d.height);
252 
253         // Puis on dessine l'ensemble des figures
254         if (drawingModel ≠ null)
255         {
256             /*
257              * Application d'un Consumer<Figure> en tant que lambda expression
258              * sur le flux (éventuellement filtré) des figures permettant
259              * de dessiner les figures
260              */
261             drawingModel.stream().forEach((Figure f) → f.draw(g2D));
262         }
263     }
264 
265     /**
266      * Mise en place d'un nouveau creationListener
267      *
268      * @param cl le nouveau creationListener
269      */
270     public void addCreationListener(AbstractCreationListener cl)
271     {
272         if (cl ≠ null)
273         {
274             addMouseListener(cl);
275             addMouseMotionListener(cl);
276             // System.out.println("CreationListener " + cl + " added");
277         }
278         else
279         {
280             System.err.println("DrawingPanel.setCreationListener(null)");
281         }
282     }
283 
284     /**
285      * Retrait d'un creationListenener
286      *
287      * @param cl le creationListener à retirer
288      */
289     public void removeCreationListener(AbstractCreationListener cl)
290     {
291         if (cl ≠ null)
292         {
293             removeMouseListener(cl);
294             removeMouseMotionListener(cl);
295             // System.out.println("CreationListener " + cl + " removed");
296         }
297     }
298 
299     /**
300      * Mise à jour déclenchée par un {@link Observable#notifyObservers()} : en
301      * l'occurence le modèle de dessin ({@link Drawing}) lorsque celui ci est
302      * modifié. Cette mise à jour déclenche une requête de redessin du panel.
303      *
304      * @param observable l'observable avant déclenché cette MAJ
305      * @param data les données (evt) transmises par l'observable
306      * @see java.util.Observer#update(java.util.Observable, java.lang.Object)
307      */
308     @Override
309     public void update(Observable observable, Object data)
310     {
311         if (observable instanceof Drawing)
312         {
313             // Le modèle à changé il faut redessiner les figures
314             repaint();
315         }
316     }
317 
318     /**
319      * Rafraichissement des panneaux d'information lors du déplacement de la
320      * souris
321      *
322      * @param e l'évènement souris associé
323      */
324     @Override
325     public void mouseDragged(MouseEvent e)
326     {
327         // Déplacement de la souris (btn enfoncé) : MAJ des coordonnées
328         // de la souris dans le coordLabel et infoPanel
```



```
329         refreshCoordLabel(e.getPoint());
330         refreshInfoPanel(e.getPoint());
331     }
332 
333     /**
334      * Rafraichissement des panneaux d'information lors du déplacement (bouton
335      * enfoncé) de la souris
336      *
337      * @param e l'évènement souris associé
338      */
339     @Override
340     public void mouseMoved(MouseEvent e)
341     {
342         // Déplacement de la souris : MAJ des coordonnées
343         // de la souris dans le coordLabel et infoPanel
344         refreshCoordLabel(e.getPoint());
345         refreshInfoPanel(e.getPoint());
346     }
347 
348     @Override
349     public void mouseClicked(MouseEvent e)
350     {
351         // Rien
352     }
353 
354     /**
355      * Reprise du rafraichissement des panneaux d'information lorsque la souris
356      * rentre dans ce panel.
357      *
358      * @param e l'évènement souris associé
359      */
360     @Override
361     public void mouseEntered(MouseEvent e)
362     {
363         sendInfoState = true;
364         refreshCoordLabel(e.getPoint());
365         refreshInfoPanel(e.getPoint());
366     }
367 
368     /**
369      * Arrêt du rafraichissement des panneaux d'information et effacement de ces
370      * panneaux lorsque la souris sort du panel.
371      *
372      * @param e l'évènement souris associé
373      */
374     @Override
375     public void mouseExited(MouseEvent e)
376     {
377         // Rien si ce n'est de remettre les coordonnés dans la barre d'état
378         // à x = ___ y = ___
379         sendInfoState = false;
380         refreshCoordLabel(e.getPoint());
381         infoPanel.resetLabels();
382     }
383 
384     @Override
385     public void mousePressed(MouseEvent e)
386     {
387         // Rien
388     }
389 
390     @Override
391     public void mouseReleased(MouseEvent e)
392     {
393         // Rien
394     }
395 
396     /**
397      * Rafraichissement du {@link #coordLabel} (s'il est non null) avec de
398      * nouvelles coordonnées ou bien avec la {@link #defaultCoordString} si l'on
399      * affiche pas les coordonnées
400      *
401      * @param x l'abcisse des coordonnées à afficher
402      * @param y l'ordonnée des coordonnées à afficher
403      */
404     private void refreshCoordLabel(Point p)
405     {
406         if ((coordLabel ≠ null) ∧ (p ≠ null))
407         {
408             if (sendInfoState)
409             {
410                 String xs = coordFormat.format(p.getX());
```

```
411                 String ys = coordFormat.format(p.getY());
412                 coordLabel.setText("x:" + xs + "y:" + ys);
413             }
414             else
415             {
416                 coordLabel.setText(defaultCoordString);
417             }
418         }
419     }
420
421     /**
422      * Rafraichissement du panneau d'information {@link #infoPanel}
423      *
424      * @param p la position du curseur pour déclencher la recherche de figures
425      *              sous ce curseur
426      */
427     private void refreshInfoPanel(Point2D p)
428     {
429         if ((infoPanel ≠ null) ∧ sendInfoState)
430         {
431             Figure selectedFigure = drawingModel.getFigureAt(p);
432
433             if (selectedFigure ≠ null)
434             {
435                 infoPanel.updateLabels(selectedFigure);
436             }
437             else
438             {
439                 infoPanel.resetLabels();
440             }
441         }
442     }
443 }
```



```
1   package widgets;
2
3   import java.awt.BorderLayout;
4   import java.awt.Color;
5   import java.awt.Component;
6   import java.awt.Dimension;
7   import java.awt.HeadlessException;
8   import java.awt.Paint;
9   import java.awt.Toolkit;
10  import java.awt.event.ActionEvent;
11  import java.awt.event.ActionListener;
12  import java.awt.event.InputEvent;
13  import java.awt.event.ItemEvent;
14  import java.awt.event.ItemListener;
15  import java.awt.event.KeyEvent;
16  import java.util.EventObject;
17
18  import javax.swing.AbstractAction;
19  import javax.swing.AbstractButton;
20  import javax.swing.Action;
21  import javax.swing.Box;
22  import javax.swing.BoxLayout;
23  import javax.swing.JButton;
24  import javax.swing.JCheckBoxMenuItem;
25  import javax.swing.JColorChooser;
26  import javax.swing.JComboBox;
27  import javax.swing.JFrame;
28  import javax.swing.JLabel;
29  import javax.swing.JMenu;
30  import javax.swing.JMenuBar;
31  import javax.swing.JMenuItem;
32  import javax.swing.JOptionPane;
33  import javax.swing.JPanel;
34  import javax.swing.JScrollPane;
35  import javax.swing.JSeparator;
36  import javax.swing.JSpinner;
37  import javax.swing.JToolBar;
38  import javax.swing.KeyStroke;
39  import javax.swing.SpinnerNumberModel;
40  import javax.swing.border.BevelBorder;
41  import javax.swing.event.ChangeEvent;
42  import javax.swing.event.ChangeListener;
43
44  import figures.Drawing;
45  import figures.creationListeners.AbstractCreationListener;
46  import figures.enums.FigureType;
47  import figures.enums.LineType;
48  import figures.enums.PaintToType;
49  import filters.EdgeColorFilter;
50  import filters.FillColorFilter;
51  import filters.LineFilter;
52  import filters.ShapeFilter;
53  import utils.IconFactory;
54  import utils.PaintFactory;
55
56  /**
57   * Classe de la fenêtre principale de l'éditeur de figures
58   * @author davidroussel
59   */
60  public class EditorFrame extends JFrame
61  {
62      /**
63       * Le nom de l'éditeur
64       */
65      protected static final String EditorName = "Figure Editor v3.0";
66
67      /**
68       * Le modèle de dessin sous-jacent;
69       */
70      protected Drawing drawingModel;
71
72      /**
73       * La zone de dessin dans laquelle seront dessinées les figures.
74       * On a besoin d'une référence à la zone de dessin (contrairement aux
75       * autres widgets) car il faut lui affecter un xxxCreationListener en
76       * fonction de la figure choisie dans la liste des figures possibles.
77       */
78      protected DrawingPanel drawingPanel;
79
80      /**
81       * Le creationListener à mettre en place dans le drawingPanel en fonction
82       * du type de figure choisie;
```

```
83      */
84     protected AbstractCreationListener creationListener;
85
86     /**
87      * Le label dans la barre d'état en bas dans lequel on affiche les
88      * conseils utilisateur pour créer une figure
89      */
90     protected JLabel infoLabel;
91
92     /**
93      * L'index de l'élément sélectionné par défaut pour le type de figure
94      */
95     private final static int defaultFigureTypeIndex = 0;
96
97     /**
98      * Les noms des couleurs de remplissage à utiliser pour remplir
99      * la [labeled]combobox des couleurs de remplissage
100     */
101    protected final static String[] fillColorNames = {
102        "Black",
103        "White",
104        "Red",
105        "Orange",
106        "Yellow",
107        "Green",
108        "Cyan",
109        "Blue",
110        "Magenta",
111        "Others",
112        "None"
113    };
114
115    /**
116     * Les couleurs de remplissage à utiliser en fonction de l'élément
117     * sélectionné dans la [labeled]combobox des couleurs de remplissage
118     */
119    protected final static Paint[] fillPaints = {
120        Color.black,
121        Color.white,
122        Color.red,
123        Color.orange,
124        Color.yellow,
125        Color.green,
126        Color.cyan,
127        Color.blue,
128        Color.magenta,
129        null, // Color selected by a JColorChooser
130        null // No Color
131    };
132
133    /**
134     * L'index de l'élément sélectionné par défaut dans les couleurs de
135     * remplissage
136     */
137    private final static int defaultFillColorIndex = 0; // black
138
139    /**
140     * L'index de la couleur de remplissage à choisir avec un
141     * {@link JColorChooser} fournit par la {@link PaintFactory}
142     */
143    private final static int specialFillColorIndex = 9;
144
145    /**
146     * Les noms des couleurs de trait à utiliser pour remplir
147     * la [labeled]combobox des couleurs de trait
148     */
149    protected final static String[] edgeColorNames = {
150        "Magenta",
151        "Red",
152        "Orange",
153        "Yellow",
154        "Green",
155        "Cyan",
156        "Blue",
157        "Black",
158        "Others"
159    };
160
161    /**
162     * Les couleurs de trait à utiliser en fonction de l'élément
163     * sélectionné dans la [labeled]combobox des couleurs de trait
164     */
```



```
165    protected final static Paint[] edgePaints = {
166        Color.magenta,
167        Color.red,
168        Color.orange,
169        Color.yellow,
170        Color.green,
171        Color.cyan,
172        Color.blue,
173        Color.black,
174        null // Color selected by a JColorChooser
175    };
176
177    /**
178     * L'index de l'élément sélectionné par défaut dans les couleurs de
179     * trait
180     */
181    private final static int defaultEdgeColorIndex = 6; // blue;
182
183    /**
184     * L'index de la couleur de remplissage à choisir avec un
185     * {@link JColorChooser} fournit par la {@link PaintFactory}
186     */
187    private final static int specialEdgeColorIndex = 8;
188
189    /**
190     * L'index de l'élément sélectionné par défaut dans les types de
191     * trait
192     */
193    private final static int defaultEdgeTypeIndex = 1; // solid
194
195    /**
196     * La largeur de trait par défaut
197     */
198    private final static int defaultEdgeWidth = 4;
199
200    /**
201     * Largeur de trait minimum
202     */
203    private final static int minEdgeWidth = 1;
204
205    /**
206     * Largeur de trait maximum
207     */
208    private final static int maxEdgeWidth = 30;
209
210    /**
211     * l'incrément entre deux largeurs de trait
212     */
213    private final static int stepEdgeWidth = 1;
214
215    /**
216     * Action déclenchée lorsque l'on clique sur le bouton quit ou sur l'item
217     * de menu quit
218     */
219    private final Action quitAction = new QuitAction();
220
221    /**
222     * Action déclenchée lorsque l'on clique sur le bouton undo ou sur l'item
223     * de menu undo
224     */
225    private final Action undoAction = new UndoAction();
226
227    /**
228     * Action déclenchée lorsque l'on clique sur le bouton clear ou sur l'item
229     * de menu clear
230     */
231    private final Action clearAction = new ClearAction();
232
233    /**
234     * Action déclenchée lorsque l'on clique sur le bouton about ou sur l'item
235     * de menu about
236     */
237    private final Action aboutAction = new AboutAction();
238
239    /**
240     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
241     * des cercles
242     */
243    private final Action circleFilterAction = new ShapeFilterAction(FigureType.CIRCLE);
244
245    /**
246     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
```

```
     * des ellipse
     */
    private final Action ellipseFilterAction = new ShapeFilterAction(FigureType.ELLIPSE);

    /**
     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
     * des rectangles
     */
    private final Action rectangleFilterAction = new ShapeFilterAction(FigureType.RECTANGLE);

    /**
     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
     * des rectangles arrondis
     */
    private final Action rRectangleFilterAction = new ShapeFilterAction(FigureType.ROUNDED_RECTANGLE
);

    /**
     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
     * des polygones
     */
    private final Action polyFilterAction = new ShapeFilterAction(FigureType.POLYGON);

    /**
     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
     * des type de lignes vides
     */
    private final Action noneLineFilterAction = new LineFilterAction(LineType.NONE);

    /**
     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
     * des type de lignes pleines
     */
    private final Action solidLineFilterAction = new LineFilterAction(LineType.SOLID);

    /**
     * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
     * des type de lignes pointillées
     */
    private final Action dashedLineFilterAction = new LineFilterAction(LineType.DASHED);

    /**
     * Constructeur de la fenètre de l'éditeur.
     * Construit les widgets et assigne les actions et autres listeners
     * aux widgets
     * @throws HeadlessException
     */
    public EditorFrame() throws HeadlessException
    {
        boolean isMacOS = System.getProperty("os.name").startsWith("Mac OS");

        /*
         * Construire l'interface graphique en utilisant WindowBuilder:
         * Menu Contextuel -> Open With -> WindowBuilder Editor puis
         * aller dans l'onglet Design
         */
        setPreferredSize(new Dimension(650, 450));
        drawingModel = new Drawing();
        creationListener = null;

        setTitle(EditorName);
        if (¬isMacOS)
        {
            setIconImage(Toolkit.getDefaultToolkit().getImage(
                    EditorFrame.class.getResource("/images/Logo.png")));
        }


        // ---------------------------------------------------------------
        // Toolbar en haut
        // ---------------------------------------------------------------
        JToolBar toolBar = new JToolBar();
        toolBar.setFloatable(false);
        getContentPane().add(toolBar, BorderLayout.NORTH);

        JButton btnCancel = new JButton("Undo");
        btnCancel.setAction(undoAction);
        toolBar.add(btnCancel);

        JButton btnErase = new JButton("Erase");
        btnErase.setAction(clearAction);
        toolBar.add(btnErase);
```



```

        JButton btnAbout = new JButton("About");
        btnAbout.setAction(aboutAction);
        toolBar.add(btnAbout);

        Component toolBoxSpringer = Box.createHorizontalGlue();
        toolBar.add(toolBoxSpringer);

        JButton btnClose = new JButton("Close");
        btnClose.setAction(quitAction);
        toolBar.add(btnClose);

        // ---------------------------------------------------------------
        // Barre d'état en bas
        // ---------------------------------------------------------------
        JPanel bottomPanel = new JPanel();
        bottomPanel.setBorder(new BevelBorder(BevelBorder.LOWERED, null, null,
                null, null));
        getContentPane().add(bottomPanel, BorderLayout.SOUTH);
        bottomPanel.setLayout(new BoxLayout(bottomPanel, BoxLayout.X_AXIS));

        infoLabel = new JLabel(AbstractCreationListener.defaultTip);
        bottomPanel.add(infoLabel);

        Component horizontalGlue = Box.createHorizontalGlue();
        bottomPanel.add(horizontalGlue);

        JLabel coordsLabel = new JLabel(DrawingPanel.defaultCoordString);
        bottomPanel.add(coordsLabel);

        // ---------------------------------------------------------------
        // Panneau de contrôle à gauche
        // ---------------------------------------------------------------
        JPanel leftPanel = new JPanel();
        leftPanel.setAlignmentY(Component.TOP_ALIGNMENT);
        getContentPane().add(leftPanel, BorderLayout.WEST);

        JLabeledComboBox figureTypeCombobox = new JLabeledComboBox("Shape",
                FigureType.stringValues(), defaultFigureTypeIndex,
                (ItemListener) null);
        leftPanel.setLayout(new BoxLayout(leftPanel, BoxLayout.Y_AXIS));
        leftPanel.add(figureTypeCombobox);

        JLabeledComboBox fillPaintComboBox = new JLabeledComboBox("Fill Color",
                fillColorNames, defaultFillColorIndex, (ItemListener) null);
        leftPanel.add(fillPaintComboBox);

        JLabeledComboBox edgePaintComboBox = new JLabeledComboBox("Edge Color",
                edgeColorNames, defaultEdgeColorIndex, (ItemListener) null);
        leftPanel.add(edgePaintComboBox);

        JLabeledComboBox edgeTypeCombobox = new JLabeledComboBox("Line Type",
                LineType.stringValues(), defaultEdgeTypeIndex,
                (ItemListener) null);
        leftPanel.add(edgeTypeCombobox);

        JPanel edgeWidthPanel = new JPanel();
        edgeWidthPanel.setAlignmentX(Component.LEFT_ALIGNMENT);
        leftPanel.add(edgeWidthPanel);
        edgeWidthPanel.setLayout(new BoxLayout(edgeWidthPanel, BoxLayout.X_AXIS));

        JLabel edgeWidthLabel = new JLabel("Line Width");
        edgeWidthPanel.add(edgeWidthLabel);

        JSpinner edgeWidthSpinner = new JSpinner();
        edgeWidthSpinner.setAlignmentX(Component.LEFT_ALIGNMENT);
        SpinnerNumberModel snm = new SpinnerNumberModel(defaultEdgeWidth,
                                                        minEdgeWidth,
                                                        maxEdgeWidth,
                                                        stepEdgeWidth);
        edgeWidthSpinner.setModel(snm);
        edgeWidthPanel.add(edgeWidthSpinner);

        InfoPanel infoPanel = new InfoPanel();
        infoPanel.setAlignmentX(Component.LEFT_ALIGNMENT);
        leftPanel.add(infoPanel);

        // ---------------------------------------------------------------
        // Zone de dessin
        // ---------------------------------------------------------------
        JScrollPane scrollPane = new JScrollPane();
        getContentPane().add(scrollPane, BorderLayout.CENTER);
```

```
410
411         drawingPanel = new DrawingPanel(drawingModel, coordsLabel, infoPanel);
412         scrollPane.setViewportView(drawingPanel);
413
414         // ----------------------------------------------------------------------
415         // Barre de menus
416         // ----------------------------------------------------------------------
417         JMenuBar menuBar = new JMenuBar();
418         setJMenuBar(menuBar);
419
420         JMenu mnFile = new JMenu("Drawing");
421         menuBar.add(mnFile);
422
423         JMenuItem mntmCancel = new JMenuItem("Cancel");
424         mntmCancel.setAction(undoAction);
425         mnFile.add(mntmCancel);
426
427         JMenuItem mntmClear = new JMenuItem("Clear");
428         mntmClear.setAction(clearAction);
429         mnFile.add(mntmClear);
430
431         JMenu mnFilter = new JMenu("Filter");
432         menuBar.add(mnFilter);
433
434         JCheckBoxMenuItem chckbxmntmFiltering = new JCheckBoxMenuItem("Filtering");
435         chckbxmntmFiltering.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_F, InputEvent.META_MAS
    K));
436         chckbxmntmFiltering.setSelected(drawingModel.getFiltering());
437         chckbxmntmFiltering.addActionListener(new ActionListener()
438         {
439             @Override
440             public void actionPerformed(ActionEvent event)
441             {
442                 AbstractButton button = (AbstractButton) event.getSource();
443                 boolean selected = button.getModel().isSelected();
444                 drawingModel.setFiltering(selected);
445             }
446         });
447         mnFilter.add(chckbxmntmFiltering);
448
449         JMenu mnFigures = new JMenu("Figures");
450         mnFilter.add(mnFigures);
451
452         JCheckBoxMenuItem chckbxmntmCircles = new JCheckBoxMenuItem("Circles");
453         chckbxmntmCircles.setAction(circleFilterAction);
454         mnFigures.add(chckbxmntmCircles);
455
456         JCheckBoxMenuItem chckbxmntmEllipses = new JCheckBoxMenuItem("Ellipses");
457         chckbxmntmEllipses.setAction(ellipseFilterAction);
458         mnFigures.add(chckbxmntmEllipses);
459
460         JCheckBoxMenuItem chckbxmntmRectangles = new JCheckBoxMenuItem("Rectangles");
461         chckbxmntmRectangles.setAction(rectangleFilterAction);
462         mnFigures.add(chckbxmntmRectangles);
463
464         JCheckBoxMenuItem chckbxmntmRoundedRectangles = new JCheckBoxMenuItem("Rounded Rectangles");
465         chckbxmntmRoundedRectangles.setAction(rRectangleFilterAction);
466         mnFigures.add(chckbxmntmRoundedRectangles);
467
468         JCheckBoxMenuItem chckbxmntmPolygons = new JCheckBoxMenuItem("Polygons");
469         chckbxmntmPolygons.setAction(polyFilterAction);
470         mnFigures.add(chckbxmntmPolygons);
471
472         JMenu mnColors = new JMenu("Colors");
473         mnFilter.add(mnColors);
474
475         JCheckBoxMenuItem chckbxmntmFillColor = new JCheckBoxMenuItem("Fill Color");
476         chckbxmntmFillColor.addActionListener(new ActionListener()
477         {
478             @Override
479             public void actionPerformed(ActionEvent event)
480             {
481                 AbstractButton button = (AbstractButton) event.getSource();
482                 boolean selected = button.getModel().isSelected();
483                 if (selected)
484                 {
485                     // Choose color from dialog
486                     // Paint newPaint = PaintFactory.getPaint(PaintFactory.getPaint(button.getParent
    (), "Choose Color", (Color) drawingModel.getFillpaint()));
487                     drawingModel.setFillColorFilter(new FillColorFilter(drawingModel.getFillpaint())
    );
488                 }
```



```
489                 else
490                 {
491                     drawingModel.setFillColorFilter(null);
492                 }
493             }
494         });
495         chckbxmntmFillColor.setIcon(IconFactory.getIcon("FillColor_small"));
496         mnColors.add(chckbxmntmFillColor);
497
498         JCheckBoxMenuItem chckbxmntmEdgeColor = new JCheckBoxMenuItem("Edge Color");
499         chckbxmntmEdgeColor.addActionListener(new ActionListener()
500         {
501             @Override
502             public void actionPerformed(ActionEvent event)
503             {
504                 AbstractButton button = (AbstractButton) event.getSource();
505                 boolean selected = button.getModel().isSelected();
506                 if (selected)
507                 {
508                     // Choose color from dialog
509                     // Paint newPaint = PaintFactory.getPaint(PaintFactory.getPaint(button.getParent
    (), "Choose Color", (Color) drawingModel.getEdgePaint()));
510                     drawingModel.setEdgeColorFilter(new EdgeColorFilter(drawingModel.getEdgePaint())
    );
511                 }
512                 else
513                 {
514                     drawingModel.setEdgeColorFilter(null);
515                 }
516             }
517         });
518         chckbxmntmEdgeColor.setIcon(IconFactory.getIcon("EdgeColor_small"));
519         mnColors.add(chckbxmntmEdgeColor);
520
521         JMenu mnStrokes = new JMenu("Strokes");
522         mnFilter.add(mnStrokes);
523
524         JCheckBoxMenuItem chckbxmntmNone = new JCheckBoxMenuItem("None");
525         chckbxmntmNone.setAction(noneLineFilterAction);
526         chckbxmntmNone.setIcon(IconFactory.getIcon("None_small"));
527         mnStrokes.add(chckbxmntmNone);
528
529         JCheckBoxMenuItem chckbxmntmSolid = new JCheckBoxMenuItem("Solid");
530         chckbxmntmSolid.setAction(solidLineFilterAction);
531         chckbxmntmSolid.setIcon(IconFactory.getIcon("Solid_small"));
532         mnStrokes.add(chckbxmntmSolid);
533
534         JCheckBoxMenuItem chckbxmntmDashed = new JCheckBoxMenuItem("Dashed");
535         chckbxmntmDashed.setAction(dashedLineFilterAction);
536         chckbxmntmDashed.setIcon(IconFactory.getIcon("Dashed_small"));
537         mnStrokes.add(chckbxmntmDashed);
538
539         if (¬isMacOS)
540         {
541             JSeparator separator = new JSeparator();
542             mnFile.add(separator);
543
544             JMenuItem mntmQuit = new JMenuItem("Quit");
545             mntmQuit.setAction(quitAction);
546             mnFile.add(mntmQuit);
547
548             JMenu mnHelp = new JMenu("Help");
549             menuBar.add(mnHelp);
550
551             JMenuItem mntmAbout = new JMenuItem("About...");
552             mntmAbout.setAction(aboutAction);
553             mnHelp.add(mntmAbout);
554         }
555
556
557         // ----------------------------------------------------------------------
558         // Ajout des contrôleurs aux widgets
559         // pour connaître les Listeners applicable à un widget
560         // dans WindowBuilder, sélectionnez un widger de l'UI puis Menu
561         // Conextuel -> Add event handler
562         // ----------------------------------------------------------------------
563         figureTypeCombobox.addItemListener(
564                 new ShapeItemListener(
565                         FigureType.fromInteger(
566                                 figureTypeCombobox.getSelectedIndex())));
567
568         fillPaintComboBox.addItemListener(new ColorItemListener(fillPaints,
```

```
569                 fillPaintComboBox.getSelectedIndex(), specialFillColorIndex,
570                 PaintToType.FILL));
571
572         edgePaintComboBox.addItemListener(new ColorItemListener(edgePaints,
573                 edgePaintComboBox.getSelectedIndex(), specialEdgeColorIndex,
574                 PaintToType.EDGE));
575
576         edgeTypeCombobox.addItemListener(new EdgeTypeListener(LineType
577                 .fromInteger(edgeTypeCombobox.getSelectedIndex())));
578
579         edgeWidthSpinner.addChangeListener(
580                 new EdgeWidthListener(defaultEdgeWidth));
581
582 //      if (isMacOS)
583 //      {
584 //          /*
585 //           * Ajout des about et quit handlers pour MacOS
586 //           */
587 //          Application.getApplication().setQuitHandler(
588 //                  (QuitHandler) quitAction);
589 //          Application.getApplication().setAboutHandler(
590 //                  (AboutHandler) aboutAction);
591 //      }
592     }
593
594     /**
595      * Action pour quitter l'application
596      * @author davidroussel
597      */
598     private class QuitAction extends AbstractAction // implements QuitHandler
599     {
600         /**
601          * Constructeur de l'action pour quitter l'application.
602          * Met en place le raccourci clavier, l'icône et la description
603          * de l'action
604          */
605         public QuitAction()
606         {
607             putValue(NAME, "Quit");
608             /*
609              * Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()
610              *  = InputEvent.CTRL_MASK on win/linux
611              *  = InputEvent.META_MASK on mac os
612              */
613             putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_Q,
614                     Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
615             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Quit"));
616             putValue(SMALL_ICON, IconFactory.getIcon("Quit_small"));
617             putValue(SHORT_DESCRIPTION, "Quits the application");
618         }
619
620         /**
621          * Opérations réalisées par l'action
622          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
623          *              ou d'un item de menu
624          */
625         @Override
626         public void actionPerformed(ActionEvent e)
627         {
628             doQuit();
629         }
630
631 //      /**
632 //       * Opérations réalisées par le quit handler
633 //       * @param e l'évènement de quit
634 //       * @param r la réponse au quit
635 //       */
636 //      @Override
637 //      public void handleQuitRequestWith(QuitEvent e, QuitResponse r)
638 //      {
639 //          doQuit();
640 //      }
641
642         /**
643          * Action réalisée pour quitter dans un {@link Action}
644          */
645         public void doQuit()
646         {
647             /*
648              * Action à effectuer lorsque l'action "undo" est cliquée :
649              * sortir avec un System.exit() (pas très propre, mais fonctionne)
650              */
```



```
651             System.exit(0);
652         }
653     }
654
655     /**
656      * Action réalisée pour effacer la dernière figure du dessin.
657      */
658     private class UndoAction extends AbstractAction
659     {
660         /**
661          * Constructeur de l'action effacer la dernière figure du dessin
662          * Met en place le raccourci clavier, l'icône et la description
663          * de l'action
664          */
665         public UndoAction()
666         {
667             putValue(NAME, "Undo");
668             putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_Z,
669                     Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
670             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Undo"));
671             putValue(SMALL_ICON, IconFactory.getIcon("Undo_small"));
672             putValue(SHORT_DESCRIPTION, "Undo last drawing");
673         }
674
675         /**
676          * Opérations réalisées par l'action
677          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
678          *              ou d'un item de menu
679          */
680         @Override
681         public void actionPerformed(ActionEvent e)
682         {
683             /*
684              * Action à effectuer lorsque l'action "undo" est cliquée :
685              * retirer la dernière figure dessinée
686              */
687             drawingModel.removeLastFigure();
688         }
689     }
690
691     /**
692      * Action réalisée pour effacer toutes les figures du dessin
693      */
694     private class ClearAction extends AbstractAction
695     {
696         /**
697          * Constructeur de l'action pour effacer toutes les figures du dessin
698          * Met en place le raccourci clavier, l'icône et la description
699          * de l'action
700          */
701         public ClearAction()
702         {
703             putValue(NAME, "Clear");
704             putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_D,
705                     Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
706             putValue(LARGE_ICON_KEY,IconFactory.getIcon("Delete"));
707             putValue(SMALL_ICON, IconFactory.getIcon("Delete_small"));
708             putValue(SHORT_DESCRIPTION, "Erase all drawings");
709         }
710
711         /**
712          * Opérations réalisées par l'action
713          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
714          *              ou d'un item de menu
715          */
716         @Override
717         public void actionPerformed(ActionEvent e)
718         {
719             /*
720              * Action à effectuer lorsque l'action "clear" est cliquée :
721              * Effacer toutes les figures du dessin
722              */
723             drawingModel.clear();
724         }
725     }
726
727     /**
728      * Action réalisée pour afficher la boite de dialogue "A propos ..."
729      */
730     private class AboutAction extends AbstractAction // implements AboutHandler
731     {
732         /**
```

```
733          * Constructeur de l'action pour afficher la boite de dialogue
734          * "A propos ..." Met en place le raccourci clavier, l'icône et la
735          * description de l'action
736          */
737         public AboutAction()
738         {
739             putValue(LARGE_ICON_KEY, IconFactory.getIcon("About"));
740             putValue(SMALL_ICON, IconFactory.getIcon("About_small"));
741             putValue(NAME, "About");
742             putValue(SHORT_DESCRIPTION, "App information");
743         }
744 
745         /**
746          * Opérations réalisées par l'action
747          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
748          *            ou d'un item de menu
749          */
750         @Override
751         public void actionPerformed(ActionEvent e)
752         {
753             doAbout(e);
754         }
755 
756 //      /**
757 //       * Actions réalisées par le about handler
758 //       * @param e l'évènement déclenchant le about handler
759 //       */
760 //      @Override
761 //      public void handleAbout(AboutEvent e)
762 //      {
763 //          doAbout(e);
764 //      }
765 
766         /**
767          * Action réalisée pour "A propos" dans un {@link Action}
768          * @param e l'évènement ayant déclenché l'action
769          */
770         public void doAbout(EventObject e)
771         {
772             /*
773              * Action à effectuer lorsque l'action "about" est cliquée :
774              * Ouvrir un MessageDialog (JOptionPane.showMessageDialog(...)) de
775              * type JOptionPane.INFORMATION_MESSAGE
776              */
777             Object source = e.getSource();
778             Component component = (source instanceof Component ?
779                     (Component)source : null);
780             JOptionPane.showMessageDialog(component, EditorName,
781                     "About ...", JOptionPane.INFORMATION_MESSAGE);
782 
783         }
784     }
785 
786     /**
787      * Action réalisée pour ajouter ou retirer un filtre de type de figure
788      */
789     private class ShapeFilterAction extends AbstractAction // implements AboutHandler
790     {
791         /**
792          * Le type de figure
793          */
794         private FigureType type;
795 
796         /**
797          * Constructeur de l'action pour mettre en place ou enlever un filtre
798          * pour filtrer les types de figures
799          */
800         public ShapeFilterAction(FigureType type)
801         {
802             this.type = type;
803             String name = type.toString();
804             putValue(LARGE_ICON_KEY, IconFactory.getIcon(name));
805             putValue(SMALL_ICON, IconFactory.getIcon(name + "_small"));
806             putValue(NAME, name);
807             putValue(SHORT_DESCRIPTION, "Set/unset " + name  + " filter");
808         }
809 
810         /**
811          * Opérations réalisées par l'action
812          * @param event l'évènement déclenchant l'action. Peut provenir d'un
813          * bouton ou d'un item de menu
814          */
```



```
815         @Override
816         public void actionPerformed(ActionEvent event)
817         {
818             AbstractButton button = (AbstractButton) event.getSource();
819             boolean selected = button.getModel().isSelected();
820             ShapeFilter sf = new ShapeFilter(type);
821             if (selected)
822             {
823                 drawingModel.addShapeFilter(sf);
824             }
825             else
826             {
827                 drawingModel.removeShapeFilter(sf);
828             }
829         }
830     }
831 
832     /**
833      * Action réalisée pour ajouter ou retirer un filtre de type trait de figure
834      */
835     private class LineFilterAction extends AbstractAction // implements AboutHandler
836     {
837         /**
838          * Le type de trait de la figure
839          */
840         private LineType type;
841 
842         /**
843          * Constructeur de l'action pour mettre en place ou enlever un filtre
844          * pour filtrer les types de figures
845          */
846         public LineFilterAction(LineType type)
847         {
848             this.type = type;
849             String name = type.toString();
850             putValue(LARGE_ICON_KEY, IconFactory.getIcon(name));
851             putValue(SMALL_ICON, IconFactory.getIcon(name + "_small"));
852             putValue(NAME, name);
853             putValue(SHORT_DESCRIPTION, "Set/unset " + name  + " filter");
854         }
855 
856         /**
857          * Opérations réalisées par l'action
858          * @param event l'évènement déclenchant l'action. Peut provenir d'un
859          * bouton ou d'un item de menu
860          */
861         @Override
862         public void actionPerformed(ActionEvent event)
863         {
864             AbstractButton button = (AbstractButton) event.getSource();
865             boolean selected = button.getModel().isSelected();
866             LineFilter lf = new LineFilter(type);
867             if (selected)
868             {
869                 drawingModel.addLineFilter(lf);
870             }
871             else
872             {
873                 drawingModel.removeLineFilter(lf);
874             }
875         }
876     }
877 
878     /**
879      * Contrôleur d'évènement permettant de modifier le type de figures à
880      * dessiner.
881      * @note dépends de #drawingModel et #infoLabel qui doivent être non
882      * null avant instanciation
883      */
884     private class ShapeItemListener implements ItemListener
885     {
886         /**
887          * Constructeur valué du contrôleur.
888          * Initialise le type de dessin dans {@link EditorFrame#drawingModel}
889          * et crée le {@link AbstractCreationListener} correspondant.
890          * @param initialIndex l'index du type de forme sélectionné afin de
891          * mettre en place le bon creationListener dans le
892          * {@link EditorFrame#drawingPanel}.
893          */
894         public ShapeItemListener(FigureType type)
895         {
896             // Mise en place du type de figure
```

```
897             drawingModel.setType(type);
898
899             // Mise en place du type de creationListener
900             creationListener = type.getCreationListener(drawingModel,
901                                                         infoLabel);
902             drawingPanel.addCreationListener(creationListener);
903         }
904
905         @Override
906         public void itemStateChanged(ItemEvent e)
907         {
908             JComboBox<?> items = (JComboBox<?>) e.getSource();
909             int index = items.getSelectedIndex();
910             int stateChange = e.getStateChange();
911             FigureType figureType = FigureType.fromInteger(index);
912             switch (stateChange)
913             {
914                 case ItemEvent.SELECTED:
915                     // Mise en place d'un nouveau type de figure
916                     drawingModel.setType(figureType);
917
918                     // Mise en place d'un nouveau type de creationListener
919                     // Après avoir retiré l'ancien
920                     drawingPanel.removeCreationListener(creationListener);
921                     creationListener = figureType.getCreationListener(drawingModel, infoLabel);
922                     drawingPanel.addCreationListener(creationListener);
923                     break;
924             }
925         }
926     }
927
928     /**
929      * Contrôleur d'évènements permettant de modifier la couleur du trait
930      * @note utilise #drawingModel qui doit être non null avant instanciation
931      */
932     private class ColorItemListener implements ItemListener
933     {
934         /**
935          * Ce à quoi s'applique la couleur choisie.
936          * Soit au rmplissage, soit au trait.
937          */
938         private PaintToType applyTo;
939
940         /**
941          * La dernière couleur choisie (pour le {@link JColorChooser})
942          */
943         private Color lastColor;
944
945         /**
946          * Le tableau des couleurs possibles
947          */
948         private Paint[] colors;
949
950         /**
951          * L'index de la couleur spéciale à choisir avec un {@link JColorChooser}
952          */
953         private final int customColorIndex;
954
955         /**
956          * L'index de la dernière couleur sélectionnée dans le combobox.
957          * Afin de pouvoir y revenir si jamais le {@link JColorChooser} est
958          * annulé.
959          */
960         private int lastSelectedIndex;
961
962         /**
963          * la couleur choisie
964          */
965         private Paint paint;
966
967         /**
968          * Constructeur du contrôleur d'évènements d'un combobox permettant
969          * de choisir la couleur de templissage
970          * @param colors le tableau des couleurs possibles
971          * @param selectedIndex l'index de l'élément actuellement sélectionné
972          * @param customColorIndex l'index de la couleur spéciale parmis les
973          * colors à définir à l'aide d'un {@link JColorChooser}.
974          * @param applyTo Ce à quoi s'applique la couleur (le remplissage ou
975          * bien le trait)
976          */
977         public ColorItemListener(Paint[] colors,
978                                  int selectedIndex,
```



```
979                                  int customColorIndex,
980                                  PaintToType applyTo)
981         {
982             this.colors = colors;
983             lastSelectedIndex = selectedIndex;
984             this.customColorIndex = customColorIndex;
985             this.applyTo = applyTo;
986             lastColor = (Color) colors[selectedIndex];
987             paint = colors[selectedIndex];
988
989             applyTo.applyPaintTo(paint, drawingModel);
990         }
991
992         /**
993          * Actions à réaliser lorsque l'élément sélectionné du combox change
994          * @param e l'évènement de changement d'item du combobox
995          */
996         @Override
997         public void itemStateChanged(ItemEvent e)
998         {
999             JComboBox<?> combo = (JComboBox<?>) e.getSource();
1000            int index = combo.getSelectedIndex();
1001
1002            if ((index ≥ 0) ∧ (index < colors.length))
1003            {
1004                if ((e.getStateChange() ≡ ItemEvent.SELECTED))
1005                {
1006                    // New color has been selected
1007                    if (index ≡ customColorIndex) // Custom color from chooser
1008                    {
1009                        Paint chosenColor = PaintFactory.getPaint(combo,
1010                            "Choose " + applyTo.toString() + " Color",
1011                            lastColor);
1012                        if (chosenColor ≠ null)
1013                        {
1014                            paint = chosenColor;
1015                        }
1016                        else
1017                        {
1018                            // ColorChooser has been cancelled we should go
1019                            // back to last selected index
1020                            combo.setSelectedIndex(lastSelectedIndex);
1021
1022                            // paint does not change
1023                        }
1024                    }
1025                    else // regular color
1026                    {
1027                        paint = colors[index];
1028                    }
1029
1030                    lastColor = (Color)paint;
1031                    applyTo.applyPaintTo(paint, drawingModel);
1032                }
1033                else if (e.getStateChange() ≡ ItemEvent.DESELECTED)
1034                {
1035                    // Old color has been delesected
1036                    if ((index ≥ 0 ) ∧ (index < customColorIndex))
1037                    {
1038                        lastColor = (Color)edgePaints[index];
1039                        lastSelectedIndex = index;
1040                    }
1041                }
1042            }
1043            else
1044            {
1045                System.err.println("Unknown " + applyTo.toString()
1046                        + " color index : " + index);
1047            }
1048        }
1049    }
1050
1051    /**
1052     * Contrôleur d'évènements permettant de modifier le type de trait (normal,
1053     * pointillé, sans trait)
1054     * @note utilise #drawingModel qui doit être non null avant instanciation
1055     */
1056    private class EdgeTypeListener implements ItemListener
1057    {
1058        /**
1059         * Le type de trait à mettre en place
1060         */
```

```
1061        private LineType edgeType;
1062
1063        public EdgeTypeListener(LineType type)
1064        {
1065            edgeType = type;
1066            drawingModel.setEdgeType(edgeType);
1067        }
1068
1069        @Override
1070        public void itemStateChanged(ItemEvent e)
1071        {
1072            JComboBox<?> items = (JComboBox<?>) e.getSource();
1073            int index = items.getSelectedIndex();
1074
1075            if (e.getStateChange() == ItemEvent.SELECTED)
1076            {
1077                // actions à réaliser lorsque le type de trait change
1078                LineType type = LineType.fromInteger(index);
1079                drawingModel.setEdgeType(type);
1080            }
1081        }
1082    }
1083
1084    /**
1085     * Contrôleur d'évènement permettant de modifier la taille du trait
1086     * en fonction des valeurs d'un {@link JSpinner}
1087     */
1088    private class EdgeWidthListener implements ChangeListener
1089    {
1090        /**
1091         * Constructeur du contrôleur d'évènements contrôlant l'épaisseur du
1092         * trait
1093         * @param initialValue la valeur initiale de la largeur du trait à
1094         * appliquer au dessin (EditorFrame#drawingModel)
1095         */
1096        public EdgeWidthListener(int initialValue)
1097        {
1098            drawingModel.setEdgeWidth(initialValue);
1099        }
1100
1101        /**
1102         * Actions à réaliser lorsque la valeur du spinner change
1103         * @param e l'évènement de changement de valeur du spinner
1104         */
1105        @Override
1106        public void stateChanged(ChangeEvent e)
1107        {
1108            JSpinner spinner = (JSpinner) e.getSource();
1109            SpinnerNumberModel spinnerModel =
1110                    (SpinnerNumberModel) spinner.getModel();
1111
1112            drawingModel.setEdgeWidth(spinnerModel.getNumber().floatValue());
1113        }
1114    }
1115 }
```



```
1  package widgets;
2
3  import java.awt.BasicStroke;
4  import java.awt.Color;
5  import java.awt.GridBagConstraints;
6  import java.awt.GridBagLayout;
7  import java.awt.Insets;
8  import java.awt.Paint;
9  import java.awt.geom.Point2D;
10 import java.awt.geom.Rectangle2D;
11 import java.text.DecimalFormat;
12 import java.util.HashMap;
13 import java.util.Map;
14
15 import javax.swing.ImageIcon;
16 import javax.swing.JLabel;
17 import javax.swing.JPanel;
18 import javax.swing.SwingConstants;
19 import javax.swing.border.LineBorder;
20
21 import figures.Figure;
22 import figures.enums.FigureType;
23 import figures.enums.LineType;
24 import utils.IconFactory;
25 import utils.PaintFactory;
26
27 public class InfoPanel extends JPanel
28 {
29     /**
30      * Une chaine vide pour remplir les champs lorsque la souris n'est au dessus
31      * d'aucune figure
32      */
33     private static final String emptyString = new String();
34
35     /**
36      * Une icône vide pour remplir les chanmps avec icône lorsque la souris
37      * n'est au dessus d'aucune figure
38      */
39     private static final ImageIcon emptyIcon = IconFactory.getIcon("None");
40
41     /**
42      * Le formatteur à utiliser pour formater les coordonnés
43      */
44     private final static DecimalFormat coordFormat = new DecimalFormat("000");
45
46     /**
47      * Le label contenant le nom de la figure
48      */
49     private JLabel lblFigureName;
50
51     /**
52      * Le label contenant l'icône correspondant à la figure
53      */
54     private JLabel lblTypeicon;
55
56     /**
57      * La map contenant les différentes icônes des types de figures
58      */
59     private Map<FigureType, ImageIcon> figureIcons;
60
61     /**
62      * Le label contenant l'icône de la couleur de remplissage
63      */
64     private JLabel lblFillcolor;
65
66     /**
67      * Le label contenant l'icône de la couleur du contour
68      */
69     private JLabel lblEdgecolor;
70
71     /**
72      * Map contenant les icônes relatives aux différentes couleurs (de contour
73      * ou de remplissage)
74      */
75     private Map<Paint, ImageIcon> paintIcons;
76
77     /**
78      * Le label contenant le type de contour
79      */
80     private JLabel lblStroketype;
81
82     /**
```

```
83      * Map contenant les icônes relatives au différents types de traits de
84      * contour
85      */
86     private Map<LineType, ImageIcon> lineTypeIcons;
87
88     /**
89      * Le label contenant l'abcisse du point en haut à gauche de la figure
90      */
91     private JLabel lblTlx;
92
93     /**
94      * Le label contenant l'ordonnée du point en haut à gauche de la figure
95      */
96     private JLabel lblTly;
97
98     /**
99      * Le label contenant l'abcisse du point en bas à droite de la figure
100     */
101    private JLabel lblBrx;
102
103    /**
104     * Le label contenant l'ordonnée du point en bas à droite de la figure
105     */
106    private JLabel lblBry;
107
108    /**
109     * Le label contenant la largeur de la figure
110     */
111    private JLabel lblDx;
112
113    /**
114     * Le label contenant la hauteur de la figure
115     */
116    private JLabel lblDy;
117
118    /**
119     * Le label contenant l'abcisse du barycentre de la figure
120     */
121    private JLabel lblCx;
122
123    /**
124     * Le label contenant l'ordonnée du barycentre de la figure
125     */
126    private JLabel lblCy;
127
128    /**
129     * Create the panel.
130     */
131    public InfoPanel()
132    {
133        // ----------------------------------------------------------------
134        // Initialisation des maps
135        // ----------------------------------------------------------------
136        figureIcons = new HashMap<FigureType, ImageIcon>();
137        for (int i = 0; i < FigureType.NbFigureTypes; i++)
138        {
139            FigureType type = FigureType.fromInteger(i);
140            figureIcons.put(type, IconFactory.getIcon(type.toString()));
141        }
142
143        paintIcons = new HashMap<Paint, ImageIcon>();
144        String[] colorStrings = {
145            "Black",
146            "Blue",
147            "Cyan",
148            "Green",
149            "Magenta",
150            "None",
151            "Orange",
152            "Others",
153            "Red",
154            "White",
155            "Yellow"
156        };
157
158        for (int i = 0; i < colorStrings.length; i++)
159        {
160            Paint paint = PaintFactory.getPaint(colorStrings[i]);
161            if (paint ≠ null)
162            {
163                paintIcons.put(paint, IconFactory.getIcon(colorStrings[i]));
164            }
```



```
165        }
166
167        lineTypeIcons = new HashMap<LineType, ImageIcon>();
168        for (int i = 0; i < LineType.NbLineTypes; i++)
169        {
170            LineType type = LineType.fromInteger(i);
171            lineTypeIcons.put(type,  IconFactory.getIcon(type.toString()));
172        }
173
174        // ----------------------------------------------------------------
175        // Création de l'UI
176        // ----------------------------------------------------------------
177        setBorder(new LineBorder(new Color(0, 0, 0), 1, true));
178        GridBagLayout gridBagLayout = new GridBagLayout();
179        gridBagLayout.columnWidths = new int[] {80, 60, 60};
180        gridBagLayout.rowHeights = new int[] {30, 32, 32, 32, 20, 20, 20, 20, 20};
181        gridBagLayout.columnWeights = new double[]{0.0, 0.0, 0.0};
182        gridBagLayout.rowWeights = new double[]{0.0, 0.0, 0.0, 0.0};
183        setLayout(gridBagLayout);
184
185        lblFigureName = new JLabel("Figure Name");
186        lblFigureName.setHorizontalAlignment(SwingConstants.CENTER);
187        GridBagConstraints gbc_lblFigureName = new GridBagConstraints();
188        gbc_lblFigureName.insets = new Insets(5, 5, 5, 0);
189        gbc_lblFigureName.gridwidth = 3;
190        gbc_lblFigureName.gridx = 0;
191        gbc_lblFigureName.gridy = 0;
192        add(lblFigureName, gbc_lblFigureName);
193
194        JLabel lblType = new JLabel("type");
195        GridBagConstraints gbc_lblType = new GridBagConstraints();
196        gbc_lblType.anchor = GridBagConstraints.EAST;
197        gbc_lblType.insets = new Insets(0, 0, 5, 5);
198        gbc_lblType.gridx = 0;
199        gbc_lblType.gridy = 1;
200        add(lblType, gbc_lblType);
201
202        lblTypeicon = new JLabel(IconFactory.getIcon("Polygon"));
203        lblTypeicon.setHorizontalAlignment(SwingConstants.CENTER);
204        GridBagConstraints gbc_lblTypeicon = new GridBagConstraints();
205        gbc_lblTypeicon.insets = new Insets(0, 0, 5, 0);
206        gbc_lblTypeicon.gridwidth = 2;
207        gbc_lblTypeicon.gridx = 1;
208        gbc_lblTypeicon.gridy = 1;
209        add(lblTypeicon, gbc_lblTypeicon);
210
211        JLabel lblFill = new JLabel("fill");
212        GridBagConstraints gbc_lblFill = new GridBagConstraints();
213        gbc_lblFill.anchor = GridBagConstraints.EAST;
214        gbc_lblFill.insets = new Insets(0, 0, 5, 5);
215        gbc_lblFill.gridx = 0;
216        gbc_lblFill.gridy = 2;
217        add(lblFill, gbc_lblFill);
218
219        lblFillcolor = new JLabel(IconFactory.getIcon("White"));
220        GridBagConstraints gbc_lblFillcolor = new GridBagConstraints();
221        gbc_lblFillcolor.gridwidth = 2;
222        gbc_lblFillcolor.insets = new Insets(0, 0, 5, 0);
223        gbc_lblFillcolor.gridx = 1;
224        gbc_lblFillcolor.gridy = 2;
225        add(lblFillcolor, gbc_lblFillcolor);
226
227        JLabel lblStroke = new JLabel("stroke");
228        GridBagConstraints gbc_lblStroke = new GridBagConstraints();
229        gbc_lblStroke.anchor = GridBagConstraints.EAST;
230        gbc_lblStroke.insets = new Insets(0, 0, 5, 5);
231        gbc_lblStroke.gridx = 0;
232        gbc_lblStroke.gridy = 3;
233        add(lblStroke, gbc_lblStroke);
234
235        lblEdgecolor = new JLabel(IconFactory.getIcon("Black"));
236        GridBagConstraints gbc_lblStrokecolor = new GridBagConstraints();
237        gbc_lblStrokecolor.insets = new Insets(0, 0, 5, 5);
238        gbc_lblStrokecolor.gridx = 1;
239        gbc_lblStrokecolor.gridy = 3;
240        add(lblEdgecolor, gbc_lblStrokecolor);
241
242        lblStroketype = new JLabel(IconFactory.getIcon("Solid"));
243        GridBagConstraints gbc_lblStroketype = new GridBagConstraints();
244        gbc_lblStroketype.insets = new Insets(0, 0, 5, 0);
245        gbc_lblStroketype.gridx = 2;
246        gbc_lblStroketype.gridy = 3;
```

```
247         add(lblStroketype, gbc_lblStroketype);
248
249         JLabel lblX = new JLabel("x");
250         lblX.setFont(lblX.getFont().deriveFont(lblX.getFont().getSize() - 3f));
251         GridBagConstraints gbc_lblX = new GridBagConstraints();
252         gbc_lblX.insets = new Insets(0, 0, 5, 5);
253         gbc_lblX.gridx = 1;
254         gbc_lblX.gridy = 4;
255         add(lblX, gbc_lblX);
256
257         JLabel lblY = new JLabel("y");
258         lblY.setFont(lblY.getFont().deriveFont(lblY.getFont().getSize() - 3f));
259         GridBagConstraints gbc_lblY = new GridBagConstraints();
260         gbc_lblY.insets = new Insets(0, 0, 5, 0);
261         gbc_lblY.gridx = 2;
262         gbc_lblY.gridy = 4;
263         add(lblY, gbc_lblY);
264
265         JLabel lblTopLeft = new JLabel("top left");
266         lblTopLeft.setFont(lblTopLeft.getFont().deriveFont(lblTopLeft.getFont().getSize() - 3f));
267         GridBagConstraints gbc_lblTopLeft = new GridBagConstraints();
268         gbc_lblTopLeft.anchor = GridBagConstraints.EAST;
269         gbc_lblTopLeft.insets = new Insets(0, 0, 5, 5);
270         gbc_lblTopLeft.gridx = 0;
271         gbc_lblTopLeft.gridy = 5;
272         add(lblTopLeft, gbc_lblTopLeft);
273
274         lblTlx = new JLabel("tlx");
275         lblTlx.setFont(lblTlx.getFont().deriveFont(lblTlx.getFont().getSize() - 3f));
276         GridBagConstraints gbc_lblTlx = new GridBagConstraints();
277         gbc_lblTlx.insets = new Insets(0, 0, 5, 5);
278         gbc_lblTlx.gridx = 1;
279         gbc_lblTlx.gridy = 5;
280         add(lblTlx, gbc_lblTlx);
281
282         lblTly = new JLabel("tly");
283         lblTly.setFont(lblTly.getFont().deriveFont(lblTly.getFont().getSize() - 3f));
284         GridBagConstraints gbc_lblTly = new GridBagConstraints();
285         gbc_lblTly.insets = new Insets(0, 0, 5, 0);
286         gbc_lblTly.gridx = 2;
287         gbc_lblTly.gridy = 5;
288         add(lblTly, gbc_lblTly);
289
290         JLabel lblBottomRight = new JLabel("bottom right");
291         lblBottomRight.setFont(lblBottomRight.getFont().deriveFont(lblBottomRight.getFont().getSize(
    ) - 3f));
292         GridBagConstraints gbc_lblBottomRight = new GridBagConstraints();
293         gbc_lblBottomRight.anchor = GridBagConstraints.EAST;
294         gbc_lblBottomRight.insets = new Insets(0, 0, 5, 5);
295         gbc_lblBottomRight.gridx = 0;
296         gbc_lblBottomRight.gridy = 6;
297         add(lblBottomRight, gbc_lblBottomRight);
298
299         lblBrx = new JLabel("brx");
300         lblBrx.setFont(lblBrx.getFont().deriveFont(lblBrx.getFont().getSize() - 3f));
301         GridBagConstraints gbc_lblBrx = new GridBagConstraints();
302         gbc_lblBrx.insets = new Insets(0, 0, 5, 5);
303         gbc_lblBrx.gridx = 1;
304         gbc_lblBrx.gridy = 6;
305         add(lblBrx, gbc_lblBrx);
306
307         lblBry = new JLabel("bry");
308         lblBry.setFont(lblBry.getFont().deriveFont(lblBry.getFont().getSize() - 3f));
309         GridBagConstraints gbc_lblBry = new GridBagConstraints();
310         gbc_lblBry.insets = new Insets(0, 0, 5, 0);
311         gbc_lblBry.gridx = 2;
312         gbc_lblBry.gridy = 6;
313         add(lblBry, gbc_lblBry);
314
315         JLabel lblDimensions = new JLabel("dimensions");
316         lblDimensions.setFont(lblDimensions.getFont().deriveFont(lblDimensions.getFont().getSize() -
    3f));
317         GridBagConstraints gbc_lblDimensions = new GridBagConstraints();
318         gbc_lblDimensions.anchor = GridBagConstraints.EAST;
319         gbc_lblDimensions.insets = new Insets(0, 0, 5, 5);
320         gbc_lblDimensions.gridx = 0;
321         gbc_lblDimensions.gridy = 7;
322         add(lblDimensions, gbc_lblDimensions);
323
324         lblDx = new JLabel("dx");
325         lblDx.setFont(lblDx.getFont().deriveFont(lblDx.getFont().getSize() - 3f));
326         GridBagConstraints gbc_lblDx = new GridBagConstraints();
```



```
327         gbc_lblDx.insets = new Insets(0, 0, 5, 5);
328         gbc_lblDx.gridx = 1;
329         gbc_lblDx.gridy = 7;
330         add(lblDx, gbc_lblDx);
331
332         lblDy = new JLabel("dy");
333         lblDy.setFont(lblDy.getFont().deriveFont(lblDy.getFont().getSize() - 3f));
334         GridBagConstraints gbc_lblDy = new GridBagConstraints();
335         gbc_lblDy.insets = new Insets(0, 0, 5, 0);
336         gbc_lblDy.gridx = 2;
337         gbc_lblDy.gridy = 7;
338         add(lblDy, gbc_lblDy);
339
340         JLabel lblCenter = new JLabel("center");
341         lblCenter.setFont(lblCenter.getFont().deriveFont(lblCenter.getFont().getSize() - 3f));
342         GridBagConstraints gbc_lblCenter = new GridBagConstraints();
343         gbc_lblCenter.anchor = GridBagConstraints.EAST;
344         gbc_lblCenter.insets = new Insets(0, 0, 0, 5);
345         gbc_lblCenter.gridx = 0;
346         gbc_lblCenter.gridy = 8;
347         add(lblCenter, gbc_lblCenter);
348
349         lblCx = new JLabel("cx");
350         lblCx.setFont(lblCx.getFont().deriveFont(lblCx.getFont().getSize() - 3f));
351         GridBagConstraints gbc_lblCx = new GridBagConstraints();
352         gbc_lblCx.insets = new Insets(0, 0, 0, 5);
353         gbc_lblCx.gridx = 1;
354         gbc_lblCx.gridy = 8;
355         add(lblCx, gbc_lblCx);
356
357         lblCy = new JLabel("cy");
358         lblCy.setFont(lblCy.getFont().deriveFont(lblCy.getFont().getSize() - 3f));
359         GridBagConstraints gbc_lblCy = new GridBagConstraints();
360         gbc_lblCy.gridx = 2;
361         gbc_lblCy.gridy = 8;
362         add(lblCy, gbc_lblCy);
363
364     }
365
366     /**
367      * Mise à jour de tous les labels avec les informations de figure
368      * @param figure la figure dont il faut extraire les informations
369      */
370     public void updateLabels(Figure figure)
371     {
372         // titre de la figure
373         lblFigureName.setText(figure.getName());
374
375         // Icône du type de figure
376         lblTypeicon.setIcon(figureIcons.get(figure.getType()));
377
378         // Icône de la couleur de remplissage
379         ImageIcon fillColorIcon = paintIcons.get(figure.getFillPaint());
380         if (fillColorIcon == null)
381         {
382             fillColorIcon = IconFactory.getIcon("Others");
383         }
384         lblFillcolor.setIcon(fillColorIcon);
385
386         // Icône de la couleur de trait
387         ImageIcon edgeColorIcon = paintIcons.get(figure.getEdgePaint());
388         if (edgeColorIcon == null)
389         {
390             edgeColorIcon = IconFactory.getIcon("Others");
391         }
392         lblEdgecolor.setIcon(edgeColorIcon);
393
394         // Icône du type de trait
395         BasicStroke stroke = figure.getStroke();
396         ImageIcon lineTypeIcon = null;
397         if (stroke == null)
398         {
399             lineTypeIcon = lineTypeIcons.get(LineType.NONE);
400         }
401         else
402         {
403             float[] dashArray = stroke.getDashArray();
404             if (dashArray == null)
405             {
406                 lineTypeIcon = lineTypeIcons.get(LineType.SOLID);
407             }
408             else
```

```
409             {
410                 lineTypeIcon = lineTypeIcons.get(LineType.DASHED);
411             }
412         }
413         lblStroketype.setIcon(lineTypeIcon);
414 
415         // Données numériques
416         Rectangle2D bounds = figure.getBounds2D();
417         Point2D center = figure.getCenter();
418 
419         double minX = bounds.getMinX();
420         double maxX = bounds.getMaxX();
421         double minY = bounds.getMinY();
422         double maxY = bounds.getMaxY();
423         double width = maxX - minX;
424         double height = maxY - minY;
425 
426         lblTlx.setText(coordFormat.format(minX));
427         lblTly.setText(coordFormat.format(minY));
428         lblBrx.setText(coordFormat.format(maxX));
429         lblBry.setText(coordFormat.format(maxY));
430 
431         lblDx.setText(coordFormat.format(width));
432         lblDy.setText(coordFormat.format(height));
433 
434         lblCx.setText(coordFormat.format(center.getX()));
435         lblCy.setText(coordFormat.format(center.getY()));
436     }
437 
438     /**
439      * Effacement de tous les labels
440      */
441     public void resetLabels()
442     {
443         // titre de la figure
444         lblFigureName.setText(emptyString);
445 
446         // Icône du type de figure
447         lblTypeicon.setIcon(emptyIcon);
448 
449         // Icône de la couleur de remplissage
450         lblFillcolor.setIcon(emptyIcon);
451 
452         // Icône de la couleur de trait
453         lblEdgecolor.setIcon(emptyIcon);
454 
455         // Icône du type de trait
456         lblStroketype.setIcon(emptyIcon);
457 
458         // Données numériques
459         lblTlx.setText(emptyString);
460         lblTly.setText(emptyString);
461         lblBrx.setText(emptyString);
462         lblBry.setText(emptyString);
463 
464         lblDx.setText(emptyString);
465         lblDy.setText(emptyString);
466 
467         lblCx.setText(emptyString);
468         lblCy.setText(emptyString);
469     }
470 }
```



```
1  package widgets;
2  
3  import java.awt.Component;
4  import java.awt.Dimension;
5  import java.awt.Font;
6  import java.awt.event.ItemListener;
7  
8  import javax.swing.BoxLayout;
9  import javax.swing.ImageIcon;
10 import javax.swing.JComboBox;
11 import javax.swing.JLabel;
12 import javax.swing.JList;
13 import javax.swing.JPanel;
14 import javax.swing.ListCellRenderer;
15 import javax.swing.SwingConstants;
16 
17 import utils.IconItem;
18 
19 /**
20  * Classe contenant un titre et une liste déroulante utilisant des JLabel avec
21  * des icones pour les élements de la liste déroulante
22  */
23 public class JLabeledComboBox extends JPanel
24 {
25     /** Le titre de cette liste */
26     private String title;
27 
28     /**
29      * Les textes et icônes pour les items
30      */
31     private IconItem[] items;
32 
33     /**
34      * La combobox utilisée à l'intérieur pour pouvoir ajouter des listener
35      * par la suite
36      */
37     private JComboBox<IconItem> combobox;
38 
39     /**
40      * Constructeur
41      * @param title le titre du panel
42      * @param captions les légendes des éléments de la liste
43      * @param selectedIndex l'élément sélectionné initialement
44      * @param listener le listener à appeller quand l'élement sélectionné de la
45      *                 liste change
46      * @see #createImageIcon(String)
47      */
48     public JLabeledComboBox(String title, String[] captions, int selectedIndex,
49             ItemListener listener)
50     {
51         setAlignmentX(Component.LEFT_ALIGNMENT);
52 
53         this.title = title;
54         items = new IconItem[captions.length];
55 
56         for (int i = 0; i < captions.length; i++)
57         {
58             items[i] = new IconItem(captions[i]);
59         }
60 
61         setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
62 
63         // Creates the title
64         JLabel label = new JLabel((this.title ≠ null ? this.title : "text"));
65         label.setHorizontalAlignment(SwingConstants.LEFT);
66         add(label);
67 
68         // Creates the Combobox
69         combobox = new JComboBox<IconItem>(items);
70         combobox.setAlignmentX(Component.LEFT_ALIGNMENT);
71         combobox.setEditable(false);
72         int index;
73         if ((selectedIndex < 0) ∨ (selectedIndex > captions.length))
74         {
75             index = 0;
76         }
77         else
78         {
79             index = selectedIndex;
80         }
81         combobox.setSelectedIndex(index);
82         combobox.addItemListener(listener);
```

```
83          // Mise en place du renderer pour les élements de la liste
84          JLabelRenderer renderer = new JLabelRenderer();
85          renderer.setPreferredSize(new Dimension(100, 32));
86          combobox.setRenderer(renderer);
87          // Ajout de la liste
88          add(combobox);
89      }
90
91      /**
92       * Ajout d'un nouveau listener déclenché lorsqu'un élément est sélectionné
93       * @param aListener le nouveau listener à ajouter.
94       */
95      public void addItemListener(ItemListener aListener)
96      {
97          if (combobox ≠ null)
98          {
99              combobox.addItemListener(aListener);
100         }
101     }
102
103     /**
104      * Obtention de l'index de l'élément sélectionné dans le combobox
105      * @return l'index de l'élément sélectionné dans le combobox
106      */
107     public int getSelectedIndex()
108     {
109         return combobox.getSelectedIndex();
110     }
111
112     /**
113      * Renderer pour les Labels du combobox
114      */
115     protected class JLabelRenderer extends JLabel
116         implements ListCellRenderer<IconItem>
117     {
118         /** fonte pour les items à problèmes */
119         private Font pbFont;
120
121         /**
122          * Constructeur
123          */
124         public JLabelRenderer()
125         {
126             setOpaque(true);
127             setHorizontalAlignment(LEFT);
128             setVerticalAlignment(CENTER);
129         }
130
131         /*
132          * (non-Javadoc)
133          * @see
134          * javax.swing.ListCellRenderer#getListCellRendererComponent(javax.swing
135          * .JList, java.lang.Object, int, boolean, boolean)
136          */
137         @Override
138         public Component getListCellRendererComponent(
139             JList<? extends IconItem> list, IconItem value, int index,
140             boolean isSelected, boolean cellHasFocus)
141         {
142             if (isSelected)
143             {
144                 setBackground(list.getSelectionBackground());
145                 setForeground(list.getSelectionForeground());
146             }
147             else
148             {
149                 setBackground(list.getBackground());
150                 setForeground(list.getForeground());
151             }
152
153             // Mise en place de l'icone et du texte dans le label
154             // Si l'icone est null afficher un label particulier avec
155             // setPbText
156             ImageIcon itemIcon = value.getIcon();
157             String itemString = value.getCaption();
158             setIcon(itemIcon);
159             if (itemIcon ≠ null)
160             {
161                 setText(itemString);
162                 setFont(list.getFont());
163             }
164             else
```



```
165             {
166                 setPbText(itemString + " (pas d'image)", list.getFont());
167             }
168
169             return this;
170         }
171
172         /**
173          * Mise en place du texte s'il y a un pb pour cet item
174          * @param pbText le texte à afficher
175          * @param normalFont la fonte à utiliser (italique)
176          */
177         protected void setPbText(String pbText, Font normalFont)
178         {
179             if (pbFont ≡ null)
180             { // lazily create this font
181                 pbFont = normalFont.deriveFont(Font.ITALIC);
182             }
183             setFont(pbFont);
184             setText(pbText);
185         }
186     }
187 }
```

```java
/**
 * Package contenant les différents widgets (éléments graphiques)
 */
package widgets;
```