```
# 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
```

```
83      * de ne pas exister
84      */
85     private static void macOSSettings()
86     {
87         // Remettre les menus au bon endroit (dans la barre en haut)
88         System.setProperty("apple.laf.useScreenMenuBar", "true");
89 
90 //      ImageIcon imageIcon = new ImageIcon(
91 //              Editor.class.getResource("/images/Logo.png"));
92 //      if (imageIcon.getImageLoadStatus() == MediaTracker.COMPLETE)
93 //      {
94             // Titre de l'application
95             System.setProperty(
96                     "com.apple.mrj.application.apple.menu.about.name",
97                     "Figure Editor");
98             // Chargement d'une icône pour le dock
99 //          com.apple.eawt.Application.getApplication().setDockIconImage(
100 //                  imageIcon.getImage());
101 //      }
102     }
103 }
104 
```



```
1  import java.awt.BasicStroke;
2  import java.awt.Color;
3  import java.awt.Dimension;
4  import java.awt.Font;
5  import java.awt.FontMetrics;
6  import java.awt.GradientPaint;
7  import java.awt.Graphics;
8  import java.awt.Graphics2D;
9  import java.awt.RenderingHints;
10 import java.awt.event.WindowAdapter;
11 import java.awt.event.WindowEvent;
12 import java.awt.geom.Arc2D;
13 import java.awt.geom.Ellipse2D;
14 import java.awt.geom.GeneralPath;
15 import java.awt.geom.Line2D;
16 import java.awt.geom.Path2D;
17 import java.awt.geom.Rectangle2D;
18 import java.awt.geom.RoundRectangle2D;
19 
20 import javax.swing.JApplet;
21 import javax.swing.JFrame;
22 
23 /*
24  * Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.
25  * Redistribution and use in source and binary forms, with or without
26  * modification, are permitted provided that the following conditions are met: -
27  * Redistributions of source code must retain the above copyright notice, this
28  * list of conditions and the following disclaimer. - Redistributions in binary
29  * form must reproduce the above copyright notice, this list of conditions and
30  * the following disclaimer in the documentation and/or other materials provided
31  * with the distribution. - Neither the name of Oracle or the names of its
32  * contributors may be used to endorse or promote products derived from this
33  * software without specific prior written permission. THIS SOFTWARE IS PROVIDED
34  * BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
35  * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
36  * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
37  * EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
38  * INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
39  * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
40  * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
41  * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
42  * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
43  * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
44  */
45 
46 /*
47  * This is like the FontDemo applet in volume 1, except that it uses the Java 2D
48  * APIs to define and render the graphics and text.
49  */
50 
51 public class ShapesDemo2D extends JApplet
52 {
53     protected final static int maxCharHeight = 15;
54     protected final static int minFontSize = 6;
55 
56     protected final static Color bg = Color.white;
57     protected final static Color fg = Color.black;
58     protected final static Color red = Color.red;
59     protected final static Color white = Color.white;
60 
61     protected final static BasicStroke stroke = new BasicStroke(2.0f);
62     protected final static BasicStroke wideStroke = new BasicStroke(8.0f,
63             BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
64 
65     protected final static float lastWidth = 20.0f;
66     protected final static float dash1[] = { 2*lastWidth };
67     protected final static BasicStroke dashed = new BasicStroke(1.0f,
68             BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, 10.0f, dash1, 0.0f);
69     protected final static BasicStroke fatDashed = new BasicStroke(lastWidth,
70             BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, lastWidth, dash1, 0.0f);
71     protected Dimension totalSize;
72     protected FontMetrics fontMetrics;
73 
74     @Override
75     public void init()
76     {
77         // Initialize drawing colors
78         setBackground(bg);
79         setForeground(fg);
80     }
81 
82     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.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;

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



```
247      *
248      * @param p le point où initialiser la figure
249      * @return la nouvelle figure créée à x et y avec les paramètres courants
250      */
251     public Figure initiateFigure(Point2D p)
252     {
253         Figure newFigure = null;
254 
255         /*
256          * TODO Maintenant que l'on s'apprête effectivement à créer une figure
257          * on ajoute les Paints et le Stroke aux factories
258          */
259 
260         /*
261          * TODO Obtention de la figure correspondant au type de figure choisi
262          * grâce à type.getFigure(...)
263          */
264 
265         /*
266          * TODO Ajout de la figure à #figures
267          */
268 
269         /* TODO Notification des observers */
270 
271         return newFigure;
272     }
273 
274     /**
275      * Obtention de la dernière figure (implicitement celle qui est en cours de
276      * dessin)
277      * @return la dernière figure du dessin
278      */
279     public Figure getLastFigure()
280     {
281         /*
282          * TODO Remplacer par l'implémentation
283          */
284         return null;
285     }
286 
287     /**
288      * Obtention de la dernière figure contenant le point p.
289      * @param p le point sous lequel on cherche une figure
290      * @return une référence vers la dernière figure contenant le point p ou à
291      *          défaut null.
292      */
293     public Figure getFigureAt(Point2D p)
294     {
295         /*
296          * TODO Remplacer par l'implémentation
297          */
298         return null;
299     }
300 
301     /**
302      * Retrait de la dernière figure (sera déclencé par une action undo)
303      * @post le modèle de dessin a été mis à jour
304      */
305     public void removeLastFigure()
306     {
307         // TODO Compléter ...
308     }
309 
310     /**
311      * Effacement de toutes les figures (sera déclenché par une action clear)
312      * @post le modèle de dessin a été mis à jour
313      */
314     public void clear()
315     {
316         // TODO Compléter ...
317     }
318 
319     /**
320      * Accesseur de l'état de filtrage
321      * @return l'état courant de filtrage
322      */
323     public boolean getFiltering()
324     {
325         return filtering;
326     }
327 
328     /**
```

```
329         * Changement d'état du filtrage
330         * @param filtering le nouveau statut de filtrage
331         * @post le modèle de dessin a été mis à jour
332         */
333        public void setFiltering(boolean filtering)
334        {
335            // TODO Compléter ...
336        }
337
338        /**
339         * Ajout d'un filtre pour filtrer les types de figures
340         * @param filter le filtre à ajouter
341         * @return true si le filtre n'était pas déjà présent dans l'ensemble des
342         * filtres filtrant les types de figures, false sinon
343         * @post si le filtre a été ajouté, une mise à jour est déclenchée
344         */
345        public boolean addShapeFilter(ShapeFilter filter)
346        {
347            boolean added = false;
348
349            // TODO Compléter ...
350
351            return added;
352        }
353
354        /**
355         * Retrait d'un filtre filtrant les types de figures
356         * @param filter le filtre à retirer
357         * @return true si le filtre faisait partie des filtres filtrant les types
358         * de figure et a été retiré, false sinon.
359         * @post si le filtre a éré retiré, une mise à jour est déclenchée
360         */
361        public boolean removeShapeFilter(ShapeFilter filter)
362        {
363            boolean removed = false;
364
365            // TODO Compléter ...
366
367            return removed;
368        }
369
370        /**
371         * Mise en place du filtre de couleur de remplissage
372         * @param filter le filtre de couleur de remplissage à appliquer
373         * @post le {@link #fillColorFilter} est mis en place et une mise à jour
374         * est déclenchée
375         */
376        public void setFillColorFilter(FillColorFilter filter)
377        {
378            // TODO Compléter ...
379        }
380
381        /**
382         * Mise en place du filtre de couleur de trait
383         * @param filter le filtre de couleur de trait à appliquer
384         * @post le {@link #edgeColorFilter} est mis en place et une mise à jour
385         * est déclenchée
386         */
387        public void setEdgeColorFilter(EdgeColorFilter filter)
388        {
389            // TODO Compléter ...
390        }
391
392        /**
393         * Ajout d'un filtre pour filtrer les types de ligne des figures
394         * @param filter le filtre à ajouter
395         * @return true si le filtre n'était pas déjà présent dans l'ensemble des
396         * filtres filtrant les types de lignes, false sinon
397         * @post si le filtre a été ajouté, une mise à jour est déclenchée
398         */
399        public boolean addLineFilter(LineFilter filter)
400        {
401            boolean added = false;
402
403            // TODO Compléter ...
404
405            return added;
406        }
407
408        /**
409         * Retrait d'un filtre filtrant les types de lignes
410         * @param filter le filtre à retirer
```



```
411         * @return true si le filtre faisait partie des filtres filtrant les types
412         * de lignes et a été retiré, false sinon.
413         * @post si le filtre a éré retiré, une mise à jour est déclenchée
414         */
415        public boolean removeLineFilter(LineFilter filter)
416        {
417            boolean removed = false;
418
419            // TODO Compléter ...
420
421            return removed;
422        }
423
424        /**
425         * Accès aux figures dans un stream afin que l'on puisse y appliquer
426         * de filtres
427         * @return le flux des figures éventuellement filtrés par les différents
428         * filtres
429         */
430        public Stream<Figure> stream()
431        {
432            Stream<Figure> figuresStream = figures.stream();
433
434            // TODO Compléter par LES filtrages du flux de figures
435
436            return figuresStream;
437        }
438    }
```

```java
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 Figure pour les cercles
 *
 * @author davidroussel
 * @uml.dependency supplier="java.awt.geom.Ellipse2D.Float"
 */
public class Circle extends Figure
{
    /**
     * Le rayon par défaut pour un cercle
     */
    public final static float DEFAULT_RAYON = 2.0f;

    /**
     * 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'un cercle
     *
     * @param stroke le type du trait de la bordure
     * @param edge la couleur de la bordure
     * @param fill la couleur de remplissage
     * @param center le centre du cercle
     * @param rayon le rayon du cercle
     */
    public Circle(BasicStroke stroke, Paint edge, Paint fill, Point2D center,
            float rayon)
    {
        super(stroke, edge, fill);
        instanceNumber = ++counter;
        float width = rayon * 2.0f;
        float height = width;
        float x = (float) (center.getX() - rayon);
        float y = (float) (center.getY() - rayon);
        shape = new Ellipse2D.Float(x, y, width, height);

        // System.out.println("Cercle created");
    }

    /**
     * Déplacement du dernier point de la ligne (utilisé lors du dessin d'un
     * cercle pour faire varier le centre et le rayon tant que l'on déplace un
     * point)
     *
     * @param p la nouvelle position du dernier point
     * @see figures.Figure#setLastPoint(Point2D)
     */
    @Override
    public void setLastPoint(Point2D p)
    {
        Ellipse2D.Float ellipse = (Ellipse2D.Float) shape;
        float newWidth = (float) (p.getX() - ellipse.x);
        float newHeight = (float) (p.getY() - ellipse.y);
        float size = (Math.abs(newWidth) < Math.abs(newHeight) ? newWidth
                : newHeight);
        ellipse.width = size;
        ellipse.height = size;
    }

    /**
     * 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.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;
    }
}
```

```java
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()
```



```java
    {
        return FigureType.ELLIPSE;
    }

}
```

```java
package figures;

import java.awt.BasicStroke;
import java.awt.Paint;
import java.awt.geom.Point2D;
import java.awt.geom.Rectangle2D;
import java.awt.geom.RectangularShape;

import figures.enums.FigureType;

/**
 * Classe de Rectangle pour les {@link Figure}
 *
 * @author davidroussel
 */
public class Rectangle extends Figure
{
    /**
     * Le compteur d'instance des cercles.
     * Utilisé pour donner un numéro d'instance après l'avoir incrémenté
     */
    private static int counter = 0;

    /**
     * Création d'un rectangle 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 Rectangle(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 Rectangle2D.Float(x, y, w, h);

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

    /**
     * Création d'un rectangle sans points (utilisé dans les classes filles
     * pour initialiser seulement les couleur et le style de trait sans
     * initialiser {@link #shape}.
     *
     * @param stroke le type de trait
     * @param edge la couleur du trait
     * @param fill la couleur de remplissage
     */
    protected Rectangle(BasicStroke stroke, Paint edge, Paint fill)
    {
        super(stroke, edge, fill);

        shape = null;
    }

    /**
     * Déplacement du point en bas à droite du rectangle à la position
     * du point p
     *
     * @param p la nouvelle position du dernier point
     * @see figures.Figure#setLastPoint(Point2D)
     */
    @Override
    public void setLastPoint(Point2D p)
    {
        if (shape ≠ null)
        {
            Rectangle2D.Float rect = (Rectangle2D.Float) shape;
            float newWidth = (float) (p.getX() - rect.x);
            float newHeight = (float) (p.getY() - rect.y);
            rect.width = newWidth;
            rect.height = newHeight;
        }
    }
```



```java
    }

    /**
     * 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;
    }
}
```

```java
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 focntion 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;
    }
}
```

```java
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]);
            }
        }
    }
```



```java

    /**
     * Déplacement du dernier point du polygone
     * @param p la position du dernier point
     * @see lines.AbstractLine#setLastPoint(Point2D)
     */
    @Override
    public void setLastPoint(Point2D p)
    {
        java.awt.Polygon poly = (java.awt.Polygon) shape;
        int lastIndex = poly.npoints-1;
        if (lastIndex ≥ 0)
        {
            poly.xpoints[lastIndex] = Double.valueOf(p.getX()).intValue();
            poly.ypoints[lastIndex] = Double.valueOf(p.getY()).intValue();
        }
    }

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

        float xm = 0.0f;
        float ym = 0.0f;

        if (poly.npoints > 0)
        {
            for (int i = 0; i < poly.npoints; i++)
            {
                xm += poly.xpoints[i];
                ym += poly.ypoints[i];
            }

            xm /= poly.npoints;
            ym /= poly.npoints;
        }

        return new Point2D.Float(xm, ym);
    }

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

```java
package figures.creationListeners;

import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.geom.Point2D;

import javax.swing.JLabel;

import figures.Figure;
import figures.Drawing;

/**
 * Listener (incomplet) des évènements souris pour créer une figure. Chaque
 * figure (Cercle, Ellipse, Rectangle, etc) est graphiquement construite par une
 * suite de pressed/drag/release ou de clicks qui peut être différente pour
 * chaque type de figure. Aussi les classes filles devront implémenter leur
 * propre xxxCreationListener assurant la gestion de la création d'une nouvelle
 * figure.
 *
 * @author davidroussel
 */
public abstract class AbstractCreationListener implements MouseListener,
        MouseMotionListener
{
    /**
     * Le drawing model à modifier par ce creationListener. Celui ci contient
     * tous les élements nécessaires à la modification du dessin par les
     * évènements souris.
     */
    protected Drawing drawingModel;

    /**
     * La figure en cours de dessin. Obtenue avec
     * {@link Drawing#initiateFigure(java.awt.geom.Point2D)}. Evite d'avoir à
     * appeler {@link Drawing#getLastFigure()} à chaque fois que la figure en
     * cours de construction est modifiée.
     */
    protected Figure currentFigure;

    /**
     * Le label dans lequel afficher les instructions nécessaires à la
     * complétion de la figure
     */
    protected JLabel tipLabel;

    /**
     * Le point de départ de la création de la figure. Utilisé pour comparer le
     * point de départ et le point terminal pour élminier les figures de taille
     * 0;
     */
    protected Point2D startPoint;

    /**
     * Le point terminal de la création de la figure. Utilisé pour comparer le
     * point de départ et le point terminal pour élminier les figures de taille
     * 0;
     */
    protected Point2D endPoint;

    /**
     * le conseil par défaut à afficher dans le {@link #tipLabel}
     */
    public static final String defaultTip = new String(
            "Cliquez pour initier une figure");

    /**
     * Le tableau de chaines de caractères contenant les conseils à
     * l'utilisateur pour chacune des étapes de la création. Par exemple [0] :
     * cliquez et maintenez enfoncé pour initier la figure [1] : relâchez pour
     * terminer la figure
     */
    protected String[] tips;

    /**
     * Le nombre d'étapes (typiquement click->drag->release) nécessaires à la
     * création de la figure
     */
    protected final int nbSteps;

    /**
     * L'étape actuelle de création de la figure
```



```java
     */
    protected int currentStep;

    /**
     * Constructeur protégé (destiné à être utilisé par les classes filles)
     *
     * @param model le modèle de dessin à modifier par ce creationListener
     * @param infoLabel le label dans lequel afficher les conseils d'utilisation
     * @param nbSteps le nombres d'étapes de création de la figure
     */
    protected AbstractCreationListener(Drawing model, JLabel infoLabel,
            int nbSteps)
    {
        drawingModel = model;
        currentFigure = null;
        tipLabel = infoLabel;
        this.nbSteps = nbSteps;
        currentStep = 0;

        // Allocation du nombres de conseils utilisateurs nécessaires
        tips = new String[(nbSteps > 0 ? nbSteps : 0)];

        if (drawingModel == null)
        {
            System.err.println("AbstractCreationListener caution null "
                    + "drawing model");
        }

        if (tipLabel == null)
        {
            System.err.println("AbstractCreationListener caution null "
                    + "tip label");
        }
    }

//  /**
//   * Mise en place du label dans lequel afficher les conseils d'utilisation
//   *
//   * @param label le label dans lequel afficher les conseils d'utilisation
//   */
//  public void setTipLabel(JLabel label)
//  {
//      tipLabel = label;
//  }

    /**
     * Initialisation de la création d'une nouvelle figure, détermine le point
     * de départ de la figure ({@link #startPoint}), initie une nouvelle figure
     * à la position de l'évènement ({@link Drawing#initiateFigure(Point2D)}),
     * met à jour le dessin {@link Drawing#update()}, puis passe à l'étape
     * suivante en mettant à jour les conseils utilisateurs (
     * {@link #updateTip()}). Pour la plupart des figures la création commence
     * par un appui sur le bouton gauche de la souris. A utiliser dans
     * {@link MouseListener#mousePressed(MouseEvent)} ou bien dans
     * {@link MouseListener#mouseClicked(MouseEvent)} suivant la figure à créer.
     *
     * @param e l'évènement souris à utiliser pour initier la création d'une
     *            nouvelle figure à la position de cet évènement
     */
    public void startFigure(MouseEvent e)
    {
        startPoint = e.getPoint();
        currentFigure = drawingModel.initiateFigure(e.getPoint());

        nextStep();

        drawingModel.update();
    }

    /**
     * Terminaison de la création d'une figure, remet l'étape courante à 0,
     * détermine la position du point de terminaison de la figure (
     * {@link #endPoint}), vérifie que la figure ainsi terminée n'est pas de
     * taille 0 ({@link #checkZeroSizeFigure()}), puis met à jour le dessin (
     * {@link Drawing#update()}) et les conseils utilisateurs (
     * {@link #updateTip()}). A utiliser dans un
     * {@link MouseListener#mousePressed(MouseEvent)} ou bien dans un
     * {@link MouseListener#mouseClicked(MouseEvent)} suivant la figure à créer.
     *
     * @param e l'évènement souris à utiliser lors de la terminaison d'un figure
     */
    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 }
```



```
1   package figures.creationListeners;
2   
3   import java.awt.event.MouseEvent;
4   
5   import javax.swing.JLabel;
6   
7   import figures.Drawing;
8   
9   /**
10   * Listener permettant d'enchainer les actions souris pour créer des formes
11   * rectangulaires comme des rectangles ou des ellipse (evt des cercles):
12   * <ol>
13   *  <li>bouton 1 pressé et maintenu enfoncé</li>
14   *  <li>déplacement de la souris avec le bouton enfoncé</li>
15   *  <li>relachement du bouton</li>
16   * </ol>
17   * @author davidroussel
18   */
19  public class RectangularShapeCreationListener extends AbstractCreationListener
20  {
21      /**
22       * Constructeur d'un listener à deux étapes: pressed->drag->release pour
23       * toutes les figures à caractère rectangulaire (Rectangle, Ellipse, evt
24       * Cercle)
25       *
26       * @param model le modèle de dessin à modifier par ce creationListener
27       * @param tipLabel le label dans lequel afficher les conseils utilisateur
28       */
29      public RectangularShapeCreationListener(Drawing model, JLabel tipLabel)
30      {
31          super(model, tipLabel, 2);
32  
33          tips[0] = new String("Cliquez et maintenez enfoncé pour initier la figure");
34          tips[1] = new String("Relâchez pour terminer la figure");
35  
36          updateTip();
37  
38          System.out.println("RectangularShapeCreationListener created");
39      }
40  
41      /**
42       * Création d'une nouvelle figure rectangulaire de taille 0 au point de
43       * l'évènement souris, si le bouton appuyé est le bouton gauche.
44       *
45       * @param e l'évènement souris
46       * @see AbstractCreationListener#startFigure(MouseEvent)
47       * @see java.awt.event.MouseListener#mousePressed(java.awt.event.MouseEvent)
48       */
49      @Override
50      public void mousePressed(MouseEvent e)
51      {
52          /*
53           * TODO si c'est le bouton 1 qui est pressé on démarre la figure
54           */
55      }
56  
57      /**
58       * Terminaison de la nouvelle figure rectangulaire si le bouton appuyé
59       * était le bouton gauche
60       * @param e l'évènement souris
61       * @see AbstractCreationListener#endFigure(MouseEvent)
62       * @see java.awt.event.MouseListener#mouseReleased(java.awt.event.MouseEvent)
63       */
64      @Override
65      public void mouseReleased(MouseEvent e)
66      {
67          /*
68           * TODO si c'est le bouton 1 qui est relaché on termine la figure
69           */
70      }
71  
72      /* (non-Javadoc)
73       * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
74       */
75      @Override
76      public void mouseClicked(MouseEvent e)
77      {
78          // Rien
79      }
80  
81      /* (non-Javadoc)
82       * @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
```

```
83      */
84     @Override
85     public void mouseEntered(MouseEvent e)
86     {
87         // Rien
88     }
89
90     /* (non-Javadoc)
91      * @see java.awt.event.MouseListener#mouseExited(java.awt.event.MouseEvent)
92      */
93     @Override
94     public void mouseExited(MouseEvent e)
95     {
96         // Rien
97     }
98
99     /*
100     * (non-Javadoc)
101     * @see java.awt.event.MouseMotionListener#mouseMoved(java.awt.event.MouseEvent)
102     */
103    @Override
104    public void mouseMoved(MouseEvent e)
105    {
106        // Rien
107    }
108
109    /**
110     * Déplacement du point en bas à droite de la figure rectangulaire, si
111     * l'on se trouve à l'étape 1 (après initialisation de la figure) et que
112     * le bouton enfoncé est bien le bouton gauche.
113     * @see java.awt.event.MouseMotionListener#mouseDragged(java.awt.event.MouseEvent)
114     */
115    @Override
116    public void mouseDragged(MouseEvent e)
117    {
118        /*
119         * TODO
120         * Si on est à l'étape 1 : on est en train de tirer le pointeur après
121         * avoir appuyé sur le bouton 1
122         *  On déplace le dernier point de la figure
123         *  On met à jour le modèle de dessin
124         */
125    }
126}
```



```
1   package figures.creationListeners;
2
3   import java.awt.event.MouseEvent;
4
5   import javax.swing.JLabel;
6
7   import figures.Drawing;
8   import figures.RoundedRectangle;
9
10  /**
11   * Listener permettant d'enchainer les actions souris nécessaires à la création
12   * d'un Rectangle arrondi:
13   *
14   * @author davidroussel
15   */
16  public class RoundedRectangleCreationListener extends AbstractCreationListener
17  {
18
19      /**
20       * Constructeur d'un Listener pour créer un polygon en plusieurs clicks
21       *
22       * @param model le modèle de dessin à modifier
23       * @param infoLabel le label dans lequel afficher les consseils utilisateurs
24       */
25      public RoundedRectangleCreationListener(Drawing model, JLabel infoLabel)
26      {
27          super(model, infoLabel, 3);
28
29          tips[0] = new String("Bouton gauche + drag pour commencer le regtangle");
30          tips[1] = new String("Relâchez pour terminer le rectangle");
31          tips[2] = new String("Clic gauche pour terminer l'arrondi du rectangle");
32
33          updateTip();
34
35          System.out.println("RoundedRectangleCreationListener created");
36      }
37
38      /**
39       * Initiation de la création d'un rectangle arrondi et passage à l'étape
40       * suivante
41       *
42       * @param e l'évènement souris associé
43       * @see figures.creationListeners.AbstractCreationListener#mousePressed(java.awt.event.MouseEvent)
44       */
45      @Override
46      public void mousePressed(MouseEvent e)
47      {
48          if ((e.getButton() ≡ MouseEvent.BUTTON1) ∧ (currentStep ≡ 0))
49          {
50              startFigure(e);
51          }
52      }
53
54      /**
55       * Terminaison de la partie rectangle du rectangle arrondi et passage à
56       * l'étape suivante
57       *
58       * @param e l'évènement souris associé
59       * @see figures.creationListeners.AbstractCreationListener#mouseReleased(java.awt.event.MouseEvent)
60       */
61      @Override
62      public void mouseReleased(MouseEvent e)
63      {
64          // Terminaison du rectangle mais pas encore de l'arrondi
65          if ((e.getButton() ≡ MouseEvent.BUTTON1) ∧ (currentStep ≡ 1))
66          {
67              nextStep();
68          }
69      }
70
71      /**
72       * Après la partie terminaison de la partie rectangle, terminaison de la
73       * partie arc (arrondi)
74       *
75       * @param e l'évènement souris associé
76       * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
77       */
78      @Override
79      public void mouseClicked(MouseEvent e)
80      {
```

```
        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();
        }
    }
}
```



```
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
    }
```



```

    /**
     * Déplacement du dernier point du polygon si l'on se trouve à la seconde
     * étape de la création
     * @param e L'évènement souris associé
     * @see
     * java.awt.event.MouseMotionListener#mouseMoved(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseMoved(MouseEvent e)
    {
        /*
         * déplacement du dernier point du polygone si l'on est toujours en
         * cours de création du polygone, rien sinon
         */
        if (currentStep > 0)
        {
            // AbstractFigure figure = drawingModel.getLastFigure();
            Figure figure = currentFigure;
            if (figure != null)
            {
                figure.setLastPoint(e.getPoint());
            }
            drawingModel.update();
        }
    }
}
```

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



```java
package figures.enums;

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

import javax.swing.JLabel;

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

/**
 * 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 null;
                // TODO return new Circle(...);
            case ELLIPSE:
                return null;
                // TODO return new Ellipse(...);
            case RECTANGLE:
                return null;
                // TODO return new Rectangle(...);
            case ROUNDED_RECTANGLE:
                return null;
                // TODO return new RoundedRectangle();
            case POLYGON:
                return null;
                // TODO return new Polygon(...);
            case NONE:
                return null;
        }

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

    /**
     * Obtention d'un CreationListener adequat pour la valeur de cet enum
     *
     * @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:
                // TODO return new FancyCreationListener(...)
            case POLYGON:
                // TODO return new AwsomeCreationListener(...);
            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)
    {
        switch (i)
        {
            case 0:
                return CIRCLE;
```



```
            case 1:
                return ELLIPSE;
            case 2:
                return RECTANGLE;
            case 3:
                return ROUNDED_RECTANGLE;
            case 4:
                return POLYGON;
            default:
                return POLYGON;
        }
    }
}
```

```java
package figures.enums;

import java.awt.BasicStroke;

/**
 * Le type de trait des lignes (continu, pointillé, ou sans trait)
 * @author davidroussel
 */
public enum LineType
{
    /**
     * Pas de trait
     */
    NONE,
    /**
     * Trait plein
     */
    SOLID,
    /**
     * Trait pointillé
     */
    DASHED;

    /**
     * Le nombre de type de lignes (à changer si l'on rajoute un type de ligne)
     */
    public static final int NbLineTypes = 3;

    /**
     * Conversion d'un entier vers un {@link LineType}.
     * A utiliser pour convertir l'index de l'élément sélectionné d'un combobox
     * dans le type de ligne correspondant
     * @param i l'entier à convertir
     * @return le LineType correspondant
     */
    public static LineType fromInteger(int i)
    {
        switch (i)
        {
            case 0:
                return NONE;
            case 1:
                return SOLID;
            case 2:
                return DASHED;
            default:
                return NONE;
        }
    }

    /**
     * Conversion d'un {@link BasicStroke} en type de ligne
     * @param stroke le stroke à examiner
     * @return le type de ligne correspondant (NONE si le stroke est nul,
     * SOLID si le stroke ne contient pas de dash array, DASHED si le stroke
     * contient un dash array.
     */
    public static LineType fromStroke(BasicStroke stroke)
    {
        if (stroke == null)
        {
            return LineType.NONE;
        }
        else
        {
            float[] dashArray = stroke.getDashArray();
            if (dashArray == null)
            {
                return LineType.SOLID;
            }
            else
            {
                return LineType.DASHED;
            }
        }
    }

    /**
     * Représentation sous forme de chaine de caractères
     * @return une chaine de caractères représentant la valeur de cet enum
     */
    @Override
    public String toString() throws AssertionError
    {
        switch (this)
        {
            case NONE:
                return new String("None");
            case SOLID:
                return new String("Solid");
            case DASHED:
                return new String("Dashed");
        }

        throw new AssertionError("LineType Unknown assertion " + this);
    }

    /**
     * Obtention d'un tableau de string contenant tous les noms des types.
     * A utiliser lors de la création d'un combobox avec :
     * LineType.stringValues()
     * @return un tableau de string contenant tous les noms des types
     */
    public static String[] stringValues()
    {
        LineType[] values = LineType.values();
        String[] stringValues = new String[values.length];
        for (int i = 0; i < values.length; i++)
        {
            stringValues[i] = values[i].toString();
        }

        return stringValues;
    }
}
```

29 nov. 15 17:49 **PaintToType.java** Page 1/1

```java
package figures.enums;

import java.awt.Paint;

import figures.Drawing;

/**
 * Enumeration de ce à quoi s'applique une couleur ({@link Paint}) à utiliser
 * dans le {@link widgets.EditorFrame.ColoItemListener}
 *
 * @author davidroussel
 */
public enum PaintToType
{
    /**
     * La couleur s'applique au remplissage
     */
    FILL,
    /**
     * La couleur s'applique au trait
     */
    EDGE;

    /**
     * Application d'une couleur au modèle de dessin en fonction de la valeur de
     * l'enum
     *
     * @param paint la couleur à appliquer
     * @param drawing le modèle de dessin sur lequel appliquer la couleur
     * @throws AssertionError si le type de l'enum est inconnu
     */
    public void applyPaintTo(Paint paint, Drawing drawing)
            throws AssertionError
    {
        switch (this)
        {
            case FILL:
                drawing.setFillPaint(paint);
                break;
            case EDGE:
                drawing.setEdgePaint(paint);
                break;
            default:
                throw new AssertionError(
                        "PaintApplicationType unknown assertion " + this);
        }
    }

    /**
     * Représentation sous forme de chaine de caractères
     *
     * @return une chaine de caractères représentant la valeur de cet enum
     */
    @Override
    public String toString() throws AssertionError
    {
        switch (this)
        {
            case FILL:
                return new String("Fill");
            case EDGE:
                return new String("Edge");
        }

        throw new AssertionError("PaintApplicationType Unknown assertion "
                + this);
    }

}
```

29 nov. 15 17:49 **package−info.java** Page 1/1

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

```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;
        }
```

```
83
84          if (obj instanceof FigureFilter<?>)
85          {
86              // TODO Compléter ...
87          }
88
89          return false;
90      }
91
92      /**
93       * Chaîne de caractères représentant le filtre
94       * @return une chaine de caractère représentantn le filtre
95       */
96      @Override
97      public String toString()
98      {
99          return new String(getClass().getSimpleName() + "<"
100             + (element ≠ null ? element.getClass().getSimpleName() : "null")
101             + ">(" + (element ≠ null ? element.toString() : "") + ")");
102     }
103 }
```



```
1   /**
2    *
3    */
4   package filters;
5
6   import java.util.Collection;
7   import java.util.Iterator;
8   import java.util.Vector;
9   import java.util.function.Predicate;
10
11  import figures.Figure;
12
13  /**
14   * Collection de filtres
15   * @author davidroussel
16   */
17  public class FigureFilters<T> implements Collection<FigureFilter<T>>, Predicate<Figure>
18  {
19      /**
20       * Vecteur de filtres
21       */
22      private Vector<FigureFilter<T>> filters;
23
24      /**
25       * Constructeur par défaut
26       */
27      public FigureFilters()
28      {
29          filters = new Vector<FigureFilter<T>>();
30      }
31
32      /**
33       * Test du prédicat
34       * @param f la figure à tester
35       * @return true si l'un au moins des prédicats de la collection est vrai,
36       * false sinon
37       * @see filters.FigureFilter#test(figures.Figure)
38       */
39      @Override
40      public boolean test(Figure f)
41      {
42          boolean result = false;
43
44          // TODO Compléter ...
45
46          return result;
47      }
48
49      /**
50       * Taille de la collection
51       * @return la taille de la collection
52       */
53      @Override
54      public int size()
55      {
56          return filters.size();
57      }
58
59      /**
60       * Teste si la collection est vide
61       * @return true si la collection est vide
62       */
63      @Override
64      public boolean isEmpty()
65      {
66          return filters.isEmpty();
67      }
68
69      /**
70       * Test de contenu d'un objet dans la collection de filtres
71       * @param o l'objet recherché dans la collection de filtres
72       * @return true si l'objet est contenu dans la collection de filtres
73       */
74      @Override
75      public boolean contains(Object o)
76      {
77          return filters.contains(o);
78      }
79
80      /**
81       * Itérateur de la collection de {@link FigureFilter}
82       * @return l'itérateur sur les filtres de la collection
```

```
83       */
84      @Override
85      public Iterator<FigureFilter<T>> iterator()
86      {
87          return filters.iterator();
88      }
89
90      /**
91       * Conversion en tableau d'objets
92       * @return un tableau d'objets contenant les éléments de la collection
93       */
94      @Override
95      public Object[] toArray()
96      {
97          return filters.toArray();
98      }
99
100     /**
101      * Conversion en tableau générique
102      * @param a un tableau générique spécimen
103      * @return un tableau générique contenant les éléments de la collection
104      */
105     @Override
106     @SuppressWarnings("hiding")
107     public <T> T[] toArray(T[] a)
108     {
109         return filters.toArray(a);
110     }
111
112     /**
113      * Ajout d'un nouveau filtre à la collection uniquement si celle ci ne
114      * contient pas déjà ce filtre
115      * @param filter le filtre à ajouter
116      * @return true si le filtre à ajouté n'était pas déjà présent dans la
117      * collection et qu'il a été ajouté
118      */
119     @Override
120     public boolean add(FigureFilter<T> filter)
121     {
122         if (¬contains(filter))
123         {
124             return filters.add(filter);
125         }
126         else
127         {
128             return false;
129         }
130     }
131
132     /**
133      * Retrait d'un objet de la collection
134      * @param o l'objet à retirer de la collection
135      * @return true si l'objet a été retiré de la collection
136      */
137     @Override
138     public boolean remove(Object o)
139     {
140         return filters.remove(o);
141     }
142
143     /**
144      * Test si une collection est entièrement contenue dans la collection
145      * @param c la collection à tester
146      * @return true si la collection c est entièrement contenue dans la
147      * collection
148      */
149     @Override
150     public boolean containsAll(Collection<?> c)
151     {
152         return filters.containsAll(c);
153     }
154
155     /**
156      * Ajout d'une collection de {@link FigureFilters} à la collection courante
157      * @param c la collection de {@link FigureFilter} à ajouter
158      * @return true si au moins un élément de la collection c a été ajouté
159      * à la collection courante
160      */
161     @Override
162     public boolean addAll(Collection<? extends FigureFilter<T>> c)
163     {
164         boolean added = false;
```



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

```java
/**
 *
 */
package filters;

import java.awt.Paint;

/**
 * Filtre filtrant les figures possédant une certaine couleur de trait
 * @author davidroussel
 */
public class EdgeColorFilter extends FigureFilter<Paint>
{
    // TODO Compléter ...
}
```



```java
/**
 *
 */
package filters;

import java.awt.Paint;

/**
 * Filtre filtrant les figures possédant une certaine couleur de remplissage
 * @author davidroussel
 */
public class FillColorFilter extends FigureFilter<Paint>
{
    // TODO Compléter ...
}
```

```java
package filters;

import figures.enums.LineType;

/**
 * Filtre filtrant les figures ayant un certain type de trait
 * @author davidroussel
 */
public class LineFilter extends FigureFilter<LineType>
{
    // TODO Compléter ...
}
```



```java
package filters;

import figures.enums.FigureType;

/**
 * Filtre de figure appliqué au type de figure
 * @author davidroussel
 */
public class ShapeFilter extends FigureFilter<FigureType>
{
    // TODO Compléter ...
}
```

```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 élements.
     */
    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)
    {
        /*
         * TODO Réucpération de l'élément dont la clé correspond à hash
         * s'il existe, null sinon
         */

        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)
    {
        /*
         * TODO Insertion d'un élément "element" dans la map avec la clé
         * correspondant à hash s'il n'existe pas déjà un tel élément dans la
         * map
         */
        return false;
    }

    /**
     * Obtention d'un élément (nouveau ou pas) : Lorsque l'élément demandé est
     * déjà présent dans la table on le renvoie directement sinon celui ci est
     * ajouté à la table avant d'être renvoyé
     * @param element l'élément demandé [celui ci pourra être détruit par le
     * garbage collector si il en existe déjà un équivalent dans la table]
     * @return l'élément demandé en provenance de la table
     */
    public T get(T element)
    {
```



```java
    {
        /*
         * TODO Compléter en utilisant get(hash) et si besoin put(hash, element)
         */
        return null;
    }

    /**
     * Nettoyage de tous les éléments
     */
    public void clear()
    {
        map.clear();
    }

    /**
     * Nettoyage avant destruction de la factory
     */
    @Override
    protected void finalize()
    {
        clear();
    }
}
```

```java
package utils;

import javax.swing.ImageIcon;

/**
 * Classe contenant une FlyweightFactory pour les les icônes, afin de pouvoir
 * réutiliser une même icône (chargée à partir d'un fichier image contenu dans
 * le package "images") à plusieurs endroits de l'interface graphique.
 * @author davidroussel
 */
public class IconFactory
{
    /**
     * le répertoire de base pour chercher les images
     */
    private final static String ImageBase = "/images/";

    /**
     * L'extension par défaut pour chercher les fichiers images
     */
    private final static String ImageType = ".png";

    /**
     * La factory stockant et fournissant les icônes
     */
    static private FlyweightFactory<ImageIcon> iconFactory =
        new FlyweightFactory<ImageIcon>();

    /**
     * Méthode d'obtention d'une icône pour un nom donné
     * @param name le nom de l'icône que l'on recherche
     * @return l'icône correspondant au nom demandé si un fichier avec ce nom
     * est trouvé dans le package/répertoire "images" ou bien null si aucune
     * image correspondant à ce nom n'est trouvée.
     */
    static public ImageIcon getIcon(String name)
    {
        // checks if there is an icon with this name in the "images" directory
        if (name.length() > 0)
        {
            int hash = name.hashCode();
            ImageIcon icon = iconFactory.get(hash);
            if (icon == null)
            {
                icon = new ImageIcon(IconFactory.class
                    .getResource(ImageBase + name + ImageType));
                if (icon.getImageLoadStatus() == java.awt.MediaTracker.COMPLETE)
                {
                    icon.setDescription(name);
                    iconFactory.put(hash, icon);
                }
                else
                {
                    System.err.println("IconFactory::getIcon(" + name
                        + "): could'nt find file " + ImageBase + name
                        + ImageType);
                }

                return iconFactory.get(hash);
            }
            else
            {
                return icon;
            }
        }
        else
        {
            System.err.println("IconFactory::getIcon(<EMPTY NAME>)");
        }

        return null;
    }
}
```



```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;
    }

    /**
     * Obtention d'un paint de la factory en déclenchant une boite de dialogue
     * de choix d'une couleur.
     * @param component le composant AWT à l'origine de la boite de dialogue
     * @param title le titre de la boite de dialogue
     * @param initialColor la couleur initiale de la boite de dialogue de choix
     * de couleus
     * @return
     */
    public static Paint getPaint(Component component,
                                 String title,
                                 Color initialColor)
    {
        if (component ≠ null)
        {
            Color color = JColorChooser.showDialog(component, title, initialColor);
            if (color ≠ null)
            {
                return paintFactory.get(color);
            }
        }

        return null;
    }
}
```

```java
package utils;

import java.awt.BasicStroke;

import figures.enums.LineType;

/**
 * Classe contenant une FlyweightFactory pour les {@link BasicStroke} afin de pouvoir
 * réutiliser un même {@link BasicStroke} à plusieurs endroits du programme
 * @author davidroussel
 */
public class StrokeFactory
{
    /**
     * Flyweight factory stockant tous les {@link BasicStroke} déjà requis
     */
    private static FlyweightFactory<BasicStroke> strokeFactory =
        new FlyweightFactory<BasicStroke>();

    /**
     * Obtention d'un {@link BasicStroke} de la factory
     * @param stroke le paint recherché
     * @return le stroke recherché
     */
    public static BasicStroke getStroke(BasicStroke stroke)
    {
        if (stroke ≠ null)
        {
            return strokeFactory.get(stroke);
        }

        return null;
    }
    /**
     * Obtention d'un {@link BasicStroke} à partir d'un type de trait et
     * d'une épaisseur de trait
     * @param type le type de trait (NONE, SOLID ou DASHED)
     * @param width l'épaisseur du trait
     * @return une {@link BasicStroke} correspondant au type et à l'épaisseur
     * de trait en provenance de la factory
     */
    public static BasicStroke getStroke(LineType type, float width)
    {
        switch (type)
        {
            default:
            case NONE:
                return null;
            case SOLID:
                return getStroke(new BasicStroke(width,
                                                 BasicStroke.CAP_ROUND,
                                                 BasicStroke.JOIN_ROUND));
            case DASHED:
                final float dash1[] = { 2 * width };
                return getStroke(new BasicStroke(width,
                                                 BasicStroke.CAP_ROUND,
                                                 BasicStroke.JOIN_ROUND,
                                                 width, dash1, 0.0f));
        }

    }
}
```



```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.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;

    /**
     * Le label (qq part dans la GUI) dans lequel afficher les coordonnées du
```



```java
     * pointeur de la souris
     */
    private JLabel coordLabel;

    /**
     * L'{@link InfoPanel} dans lequel afficher les informations à propos de
     * la figure sous le curseur.
     */
    private InfoPanel infoPanel;

    /**
     * Chaîne de caractère à afficher par défaut dans le {@link #coordLabel}
     */
    public final static String defaultCoordString = new String("x: ___ y: ___");

    /**
     * Le formatteur à utiliser pour formater les nombres dans le
     * {@link #coordLabel} et dans l'{@link #infoPanel}
     */
    private final static DecimalFormat coordFormat = new DecimalFormat("000");

    /**
     * état indiquant s'il faut envoyer les coordonnées de la souris ou la
     * figure au dessus de laquelle se trouve la souris. Lorsque le curseur sort
     * du widget (mouseExited) on cesse d'envoyer les coordonnées de la souris
     * et lorsqu'elle entre (mouseEntered) on recommence à envoyer les
     * coordonnées de la souris.
     */
    private boolean sendInfoState;

    /**
     * Constructeur de la zone de dessin à partir d'un modèle de dessin.
     *
     * @param drawing le modèle de dessin
     * @param coordLabel le label à mettre à jour avec les coordonnées du
     *            curseur de la souris
     * @param infoPanel le panneau d'information des figures à mettre à jour
     *            avec les informations relative à la figure située sous le
     *            curseur de la souris
     */
    public DrawingPanel(Drawing drawing, JLabel coordLabel, InfoPanel infoPanel)
    {
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
        size = new Dimension(800, 600);
        setPreferredSize(size);
        addComponentListener(new ResizeListener());

        setBackground(Color.WHITE);
        setLayout(null);
        setDoubleBuffered(true);

        drawingModel = drawing;
        if (drawing ≠ null)
        {
            drawingModel.addObserver(this);
        }
        else
        {
            System.err.println("DrawingPanel caution: null drawing");
        }

        this.coordLabel = coordLabel;

        // TODO Remise à zero du coordLabel

        this.infoPanel = infoPanel;

        // TODO Remise à zero de l'infoPanel

        // DrawingPanel est son propre listener d'évènements souris
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    /**
     * Accès à la taille effective du panel qui peut changer si celui-ci est
     * agrandi (avec la fenêtre dans lequel il est par exemple). Cette méthode
     * permet d'ajuster les scrollbars d'un container qui contiendrait ce panel
     * lorsque la taille de celui-ci change.
     *
     * @return la taille effective du panel de dessin
     * @see javax.swing.JComponent#getPreferredSize()
```

```
     */
    @Override
    public Dimension getPreferredSize()
    {
        return size;
    }

    /**
     * Mise en place du modèle de dessin. Met en place un nouveau modèle et s'il
     * est non null ajoute ce panel comme observateur du modèle
     *
     * @param drawing le modèle de dessin à mettre en place
     */
    public void setDrawing(Drawing drawing)
    {
        // retrait du précédent modèle de dessin (s'il existe)
        if (drawingModel ≠ null)
        {
            drawingModel.deleteObserver(this);
        }

        // Mise en place du nouveau modèle de dessin
        drawingModel = drawing;
        if (drawingModel ≠ null)
        {
            drawingModel.addObserver(this);
        }
    }

    /**
     * Mise en place du label dans lequel afficher les coordonnées du pointeur
     * de la souris.
     *
     * @param coordLabel le label dans lequel afficher les coordonnées du
     *            pointeur de la souris.
     */
    public void setCoordLabel(JLabel coordLabel)
    {
        this.coordLabel = coordLabel;
    }

    /**
     * Mise en place du panel d'information dans lequel afficher les infos sur
     * la figure située sous le curseur
     *
     * @param infoPanel l'{@link InfoPanel} à mettre en place
     */
    public void setInfoPanel(InfoPanel infoPanel)
    {
        this.infoPanel = infoPanel;
    }

    /**
     * Dessin du panel. Effacement ce celui-ci puis dessin des figures.
     * @param g le contexte graphique
     * @see javax.swing.JComponent#paintComponent(java.awt.Graphics)
     */
    @Override
    protected void paintComponent(Graphics g)
    {
        super.paintComponent(g); // Inutile

        // caractéristiques graphiques : mise en place de l'antialiasing
        Graphics2D g2D = (Graphics2D) g;
        g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                             RenderingHints.VALUE_ANTIALIAS_ON);

        // taille de la zone de dessin
        Dimension d = getSize();
        // on commence par effacer le fond
        g2D.setColor(getBackground());
        g2D.fillRect(0, 0, d.width, d.height);

        // Puis on dessine l'ensemble des figures
        if (drawingModel ≠ null)
        {
            /*
             * TODO Application d'un Consumer<Figure> en tant que lambda
             * expression sur le flux (éventuellement filtré) : pour chaque
             * figure du flux il faut la dessiner (avec sa méthode draw)
             */
        }
```



```
    }

    /**
     * Mise en place d'un nouveau creationListener
     *
     * @param cl le nouveau creationListener
     */
    public void addCreationListener(AbstractCreationListener cl)
    {
        if (cl ≠ null)
        {
            addMouseListener(cl);
            addMouseMotionListener(cl);
            // System.out.println("CreationListener " + cl + " added");
        }
        else
        {
            System.err.println("DrawingPanel.setCreationListener(null)");
        }
    }

    /**
     * Retrait d'un creationListenener
     *
     * @param cl le creationListener à retirer
     */
    public void removeCreationListener(AbstractCreationListener cl)
    {
        if (cl ≠ null)
        {
            removeMouseListener(cl);
            removeMouseMotionListener(cl);
            // System.out.println("CreationListener " + cl + " removed");
        }
    }

    /**
     * Mise à jour déclenchée par un {@link Observable#notifyObservers()} : en
     * l'occurence le modèle de dessin ({@link Drawing}) lorsque celui ci est
     * modifié. Cette mise à jour déclenche une requête de redessin du panel.
     *
     * @param observable l'observable ayant déclenché cette MAJ
     * @param data les données (evt) transmises par l'observable
     * @see java.util.Observer#update(java.util.Observable, java.lang.Object)
     */
    @Override
    public void update(Observable observable, Object data)
    {
        if (observable instanceof Drawing)
        {
            // Le modèle à changé il faut redessiner les figures
            repaint();
        }
    }

    /**
     * Rafraichissement des panneaux d'information lors du déplacement de la
     * souris
     *
     * @param e l'évènement souris associé
     */
    @Override
    public void mouseDragged(MouseEvent e)
    {
        /*
         * Déplacement de la souris (btn enfoncé) :
         * TODO MAJ des coordonnées de la souris dans le coordLabel
         * [Opt]MAJ de l'infoPanel si une figure se situe sous le curseur
         */
        refreshCoordLabel(e.getPoint());
        refreshInfoPanel(e.getPoint());
    }

    /**
     * Rafraichissement des panneaux d'information lors du déplacement (bouton
     * enfoncé) de la souris
     *
     * @param e l'évènement souris associé
     */
    @Override
    public void mouseMoved(MouseEvent e)
    {
```

```
329         /*
330          * Déplacement de la souris :
331          * TODO MAJ des coordonnées de la souris dans le coordLabel
332          * MAJ de l'infoPanel si une figure se situe sous le curseur
333          */
334         refreshCoordLabel(e.getPoint());
335         refreshInfoPanel(e.getPoint());
336     }
337
338     @Override
339     public void mouseClicked(MouseEvent e)
340     {
341         // Rien
342     }
343
344     /**
345      * Reprise du rafraichissement des panneaux d'information lorsque la souris
346      * rentre dans ce panel.
347      *
348      * @param e l'évènement souris associé
349      */
350     @Override
351     public void mouseEntered(MouseEvent e)
352     {
353         /*
354          * TODO On passe dans l'état où l'on doit mettre à jour des infos
355          * On rafraichit le coordLabel avec la position de e
356          * On rafraichit l'infoPanel "
357          */
358     }
359
360     /**
361      * Arrêt du rafraichissement des panneaux d'information et effacement de ces
362      * panneaux lorsque la souris sort du panel.
363      *
364      * @param e l'évènement souris associé
365      */
366     @Override
367     public void mouseExited(MouseEvent e)
368     {
369         /*
370          * TODO on est plus dans l'état où l'on met à jour les infos
371          * remettre les coordonnés dans la barre d'état à x = ___ y = ___
372          * Remettre les champs de l'infoPanel à vide
373          */
374     }
375
376     @Override
377     public void mousePressed(MouseEvent e)
378     {
379         // Rien
380     }
381
382     @Override
383     public void mouseReleased(MouseEvent e)
384     {
385         // Rien
386     }
387
388     /**
389      * Rafraichissement du {@link #coordLabel} (s'il est non null) avec de
390      * nouvelles coordonnées ou bien avec la {@link #defaultCoordString} si l'on
391      * affiche pas les coordonnées
392      *
393      * @param x l'abcisse des coordonnées à afficher
394      * @param y l'ordonnée des coordonnées à afficher
395      */
396     private void refreshCoordLabel(Point p)
397     {
398         /*
399          * TODO Si on est dans l'état où l'on met à jour les infos
400          *  Formattage des valeurs de la position de p
401          *  Mise à jour du coordLabel contenant les coordonées
402          * Sinon
403          *  On remet coordLabel à sa valeur par défaut
404          */
405     }
406
407     /**
408      * Rafraichissement du panneau d'information {@link #infoPanel}
409      *
410      * @param p la position du curseur pour déclencher la recherche de figures
```



```
411      *            sous ce curseur
412      */
413     private void refreshInfoPanel(Point2D p)
414     {
415         /*
416          * TODO Si on est dans l'état où l'on met à jour les infos
417          *  On recherche la dernière figure sous le curseur
418          *  Si on a trouvé une figure
419          *      met à jour l'info panel avec les infos de la figure
420          *  Sinon
421          *      On remet l'infoPanel à ses valeurs par défaut
422          */
423     }
424 }
```

```java
package widgets;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.HeadlessException;
import java.awt.Paint;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import java.util.EventObject;

import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JColorChooser;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JSpinner;
import javax.swing.KeyStroke;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import figures.Drawing;
import figures.creationListeners.AbstractCreationListener;
import figures.enums.FigureType;
import figures.enums.LineType;
import figures.enums.PaintToType;
import utils.IconFactory;
import utils.PaintFactory;

/**
 * Classe de la fenêtre principale de l'éditeur de figures
 * @author davidroussel
 */
public class EditorFrame extends JFrame
{
    /**
     * Le nom de l'éditeur
     */
    protected static final String EditorName = "Figure Editor v3.0";

    /**
     * Le modèle de dessin sous-jacent;
     */
    protected Drawing drawingModel;

    /**
     * La zone de dessin dans laquelle seront dessinées les figures.
     * On a besoin d'une référence à la zone de dessin (contrairement aux
     * autres widgets) car il faut lui affecter un xxxCreationListener en
     * fonction de la figure choisie dans la liste des figures possibles.
     */
    protected DrawingPanel drawingPanel;

    /**
     * Le creationListener à mettre en place dans le drawingPanel en fonction
     * du type de figure choisie;
     */
    protected AbstractCreationListener creationListener;

    /**
     * Le label dans la barre d'état en bas dans lequel on affiche les
     * conseils utilisateur pour créer une figure
     */
    protected JLabel infoLabel;

    /**
     * L'index de l'élément sélectionné par défaut pour le type de figure
     */
    private final static int defaultFigureTypeIndex = 0;

    /**
     * Les noms des couleurs de remplissage à utiliser pour remplir
     * la [labeled]combobox des couleurs de remplissage
     */
    protected final static String[] fillColorNames = {
        "Black",
        "White",
        "Red",
        "Orange",
```



```java
        "Yellow",
        "Green",
        "Cyan",
        "Blue",
        "Magenta",
        "Others",
        "None"
    };

    /**
     * Les couleurs de remplissage à utiliser en fonction de l'élément
     * sélectionné dans la [labeled]combobox des couleurs de remplissage
     */
    protected final static Paint[] fillPaints = {
        Color.black,
        Color.white,
        Color.red,
        Color.orange,
        Color.yellow,
        Color.green,
        Color.cyan,
        Color.blue,
        Color.magenta,
        null, // Color selected by a JColorChooser
        null // No Color
    };

    /**
     * L'index de l'élément sélectionné par défaut dans les couleurs de
     * remplissage
     */
    private final static int defaultFillColorIndex = 0; // black

    /**
     * L'index de la couleur de remplissage à choisir avec un
     * {@link JColorChooser} fournit par la {@link PaintFactory}
     */
    private final static int specialFillColorIndex = 9;

    /**
     * Les noms des couleurs de trait à utiliser pour remplir
     * la [labeled]combobox des couleurs de trait
     */
    protected final static String[] edgeColorNames = {
        "Magenta",
        "Red",
        "Orange",
        "Yellow",
        "Green",
        "Cyan",
        "Blue",
        "Black",
        "Others"
    };

    /**
     * Les couleurs de trait à utiliser en fonction de l'élément
     * sélectionné dans la [labeled]combobox des couleurs de trait
     */
    protected final static Paint[] edgePaints = {
        Color.magenta,
        Color.red,
        Color.orange,
        Color.yellow,
        Color.green,
        Color.cyan,
        Color.blue,
        Color.black,
        null // Color selected by a JColorChooser
    };

    /**
     * L'index de l'élément sélectionné par défaut dans les couleurs de
     * trait
     */
    private final static int defaultEdgeColorIndex = 6; // blue;

    /**
     * L'index de la couleur de remplissage à choisir avec un
     * {@link JColorChooser} fournit par la {@link PaintFactory}
     */
    private final static int specialEdgeColorIndex = 8;
```

```
165
166     /**
167      * L'index de l'élément sélectionné par défaut dans les types de
168      * trait
169      */
170     private final static int defaultEdgeTypeIndex = 1; // solid
171
172     /**
173      * La largeur de trait par défaut
174      */
175     private final static int defaultEdgeWidth = 4;
176
177     /**
178      * Largeur de trait minimum
179      */
180     private final static int minEdgeWidth = 1;
181
182     /**
183      * Largeur de trait maximum
184      */
185     private final static int maxEdgeWidth = 30;
186
187     /**
188      * l'incrément entre deux largeurs de trait
189      */
190     private final static int stepEdgeWidth = 1;
191
192     /**
193      * Action déclenchée lorsque l'on clique sur le bouton quit ou sur l'item
194      * de menu quit
195      */
196     private final Action quitAction = new QuitAction();
197
198     /**
199      * Action déclenchée lorsque l'on clique sur le bouton undo ou sur l'item
200      * de menu undo
201      */
202     private final Action undoAction = new UndoAction();
203
204     /**
205      * Action déclenchée lorsque l'on clique sur le bouton clear ou sur l'item
206      * de menu clear
207      */
208     private final Action clearAction = new ClearAction();
209
210     /**
211      * Action déclenchée lorsque l'on clique sur le bouton about ou sur l'item
212      * de menu about
213      */
214     private final Action aboutAction = new AboutAction();
215
216     /**
217      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
218      * des cercles
219      */
220     private final Action circleFilterAction = new ShapeFilterAction(FigureType.CIRCLE);
221
222     /**
223      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
224      * des ellipse
225      */
226     private final Action ellipseFilterAction = new ShapeFilterAction(FigureType.ELLIPSE);
227
228     /**
229      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
230      * des rectangles
231      */
232     private final Action rectangleFilterAction = new ShapeFilterAction(FigureType.RECTANGLE);
233
234     /**
235      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
236      * des rectangles arrondis
237      */
238     private final Action rRectangleFilterAction = new ShapeFilterAction(FigureType.ROUNDED_RECTANGLE
    );
239
240     /**
241      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
242      * des polygones
243      */
244     private final Action polyFilterAction = new ShapeFilterAction(FigureType.POLYGON);
245
```



```
246     /**
247      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
248      * des type de lignes vides
249      */
250     private final Action noneLineFilterAction = new LineFilterAction(LineType.NONE);
251
252     /**
253      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
254      * des type de lignes pleines
255      */
256     private final Action solidLineFilterAction = new LineFilterAction(LineType.SOLID);
257
258     /**
259      * Action déclenchée lorsque l'on clique sur l'item de menu de filtrage
260      * des type de lignes pointillées
261      */
262     private final Action dashedLineFilterAction = new LineFilterAction(LineType.DASHED);
263
264     /**
265      * Constructeur de la fenêtre de l'éditeur.
266      * Construit les widgets et assigne les actions et autres listeners
267      * aux widgets
268      * @throws HeadlessException
269      */
270     public EditorFrame() throws HeadlessException
271     {
272         boolean isMacOS = System.getProperty("os.name").startsWith("Mac OS");
273
274         /*
275          * Construire l'interface graphique en utilisant WindowBuilder:
276          * Menu Contextuel -> Open With -> WindowBuilder Editor puis
277          * aller dans l'onglet Design
278          */
279         setPreferredSize(new Dimension(650, 450));
280         drawingModel = new Drawing();
281         creationListener = null;
282
283         setTitle(EditorName);
284         if (¬isMacOS)
285         {
286             setIconImage(Toolkit.getDefaultToolkit().getImage(
287                     EditorFrame.class.getResource("/images/Logo.png")));
288         }
289
290
291         // ------------------------------------------------------------------
292         // Toolbar en haut
293         // ------------------------------------------------------------------
294         // TODO compléter ...
295
296         // ------------------------------------------------------------------
297         // Barre d'état en bas
298         // ------------------------------------------------------------------
299         // TODO compléter ...
300
301         // ------------------------------------------------------------------
302         // Panneau de contrôle à gauche
303         // ------------------------------------------------------------------
304         // TODO compléter ...
305
306         // ------------------------------------------------------------------
307         // Zone de dessin
308         // ------------------------------------------------------------------
309         // TODO compléter ...
310
311         // TODO drawingPanel = new DrawingPanel(...);
312         // TODO <zone de dessin>.setViewportView(drawingPanel);
313
314         // ------------------------------------------------------------------
315         // Barre de menus
316         // ------------------------------------------------------------------
317         // TODO compléter ...
318
319         // ------------------------------------------------------------------
320         // Ajout des contrôleurs aux widgets
321         // pour connaître les Listeners applicable à un widget
322         // dans WindowBuilder, sélectionnez un widger de l'UI puis Menu
323         // Conextuel -> Add event handler
324         // ------------------------------------------------------------------
325         // TODO compléter ...
326     }
327
```

```
328     /**
329      * Action pour quitter l'application
330      * @author davidroussel
331      */
332     private class QuitAction extends AbstractAction // implements QuitHandler
333     {
334         /**
335          * Constructeur de l'action pour quitter l'application.
336          * Met en place le raccourci clavier, l'icône et la description
337          * de l'action
338          */
339         public QuitAction()
340         {
341             putValue(NAME, "Quit");
342             /*
343              * Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()
344              *  = InputEvent.CTRL_MASK on win/linux
345              *  = InputEvent.META_MASK on mac os
346              */
347             putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_Q,
348                     Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
349             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Quit"));
350             putValue(SMALL_ICON, IconFactory.getIcon("Quit_small"));
351             putValue(SHORT_DESCRIPTION, "Quits the application");
352         }
353
354         /**
355          * Opérations réalisées par l'action
356          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
357          *            ou d'un item de menu
358          */
359         @Override
360         public void actionPerformed(ActionEvent e)
361         {
362             doQuit();
363         }
364
365 //      /**
366 //       * Opérations réalisées par le quit handler
367 //       * @param e l'évènement de quit
368 //       * @param r la réponse au quit
369 //       */
370 //      @Override
371 //      public void handleQuitRequestWith(QuitEvent e, QuitResponse r)
372 //      {
373 //          doQuit();
374 //      }
375
376         /**
377          * Action réalisée pour quitter dans un {@link Action}
378          */
379         public void doQuit()
380         {
381             /*
382              * Action à effectuer lorsque l'action "undo" est cliquée :
383              * sortir avec un System.exit() (pas très propre, mais fonctionne)
384              */
385             System.exit(0);
386         }
387     }
388
389     /**
390      * Action réalisée pour effacer la dernière figure du dessin.
391      */
392     private class UndoAction extends AbstractAction
393     {
394         /**
395          * Constructeur de l'action effacer la dernière figure du dessin
396          * Met en place le raccourci clavier, l'icône et la description
397          * de l'action
398          */
399         public UndoAction()
400         {
401             putValue(NAME, "Undo");
402             putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_Z,
403                     Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
404             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Undo"));
405             putValue(SMALL_ICON, IconFactory.getIcon("Undo_small"));
406             putValue(SHORT_DESCRIPTION, "Undo last drawing");
407         }
408
409         /**
```



```
410          * Opérations réalisées par l'action
411          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
412          *            ou d'un item de menu
413          */
414         @Override
415         public void actionPerformed(ActionEvent e)
416         {
417             /*
418              * TODO Action à effectuer lorsque l'action "undo" est cliquée :
419              * retirer la dernière figure dessinée
420              */
421         }
422     }
423
424     /**
425      * Action réalisée pour effacer toutes les figures du dessin
426      */
427     private class ClearAction extends AbstractAction
428     {
429         /**
430          * Constructeur de l'action pour effacer toutes les figures du dessin
431          * Met en place le raccourci clavier, l'icône et la description
432          * de l'action
433          */
434         public ClearAction()
435         {
436             putValue(NAME, "Clear");
437             putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_D,
438                     Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
439             putValue(LARGE_ICON_KEY,IconFactory.getIcon("Delete"));
440             putValue(SMALL_ICON, IconFactory.getIcon("Delete_small"));
441             putValue(SHORT_DESCRIPTION, "Erase all drawings");
442         }
443
444         /**
445          * Opérations réalisées par l'action
446          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
447          *            ou d'un item de menu
448          */
449         @Override
450         public void actionPerformed(ActionEvent e)
451         {
452             /*
453              * TODO Action à effectuer lorsque l'action "clear" est cliquée :
454              * Effacer toutes les figures du dessin
455              */
456         }
457     }
458
459     /**
460      * Action réalisée pour afficher la boite de dialogue "A propos ..."
461      */
462     private class AboutAction extends AbstractAction // implements AboutHandler
463     {
464         /**
465          * Constructeur de l'action pour afficher la boite de dialogue
466          * "A propos ..." Met en place le raccourci clavier, l'icône et la
467          * description de l'action
468          */
469         public AboutAction()
470         {
471             putValue(LARGE_ICON_KEY, IconFactory.getIcon("About"));
472             putValue(SMALL_ICON, IconFactory.getIcon("About_small"));
473             putValue(NAME, "About");
474             putValue(SHORT_DESCRIPTION, "App information");
475         }
476
477         /**
478          * Opérations réalisées par l'action
479          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
480          *            ou d'un item de menu
481          */
482         @Override
483         public void actionPerformed(ActionEvent e)
484         {
485             doAbout(e);
486         }
487
488 //      /**
489 //       * Actions réalisées par le about handler
490 //       * @param e l'évènement déclenchant le about handler
491 //       */
```

```
492 //      @Override
493 //      public void handleAbout(AboutEvent e)
494 //      {
495 //          doAbout(e);
496 //      }
497
498         /**
499          * Action réalisée pour "A propos" dans un {@link Action}
500          * @param e l'évènement ayant déclenché l'action
501          */
502         public void doAbout(EventObject e)
503         {
504             /*
505              * TODO Action à effectuer lorsque l'action "about" est cliquée :
506              * Ouvrir un MessageDialog (JOptionPane.showMessageDialog(...)) de
507              * type JOptionPane.INFORMATION_MESSAGE
508              */
509         }
510     }
511
512     /**
513      * Action réalisée pour ajouter ou retirer un filtre de type de figure
514      */
515     private class ShapeFilterAction extends AbstractAction // implements AboutHandler
516     {
517         /**
518          * Le type de figure
519          */
520         private FigureType type;
521
522         /**
523          * Constructeur de l'action pour mettre en place ou enlever un filtre
524          * pour filtrer les types de figures
525          */
526         public ShapeFilterAction(FigureType type)
527         {
528             this.type = type;
529             String name = type.toString();
530             putValue(LARGE_ICON_KEY, IconFactory.getIcon(name));
531             putValue(SMALL_ICON, IconFactory.getIcon(name + "_small"));
532             putValue(NAME, name);
533             putValue(SHORT_DESCRIPTION, "Set/unset " + name  + " filter");
534         }
535
536         /**
537          * Opérations réalisées par l'action
538          * @param event l'évènement déclenchant l'action. Peut provenir d'un
539          * bouton ou d'un item de menu
540          */
541         @Override
542         public void actionPerformed(ActionEvent event)
543         {
544             /*
545              * TODO Si l'AbstractButton de la source est sélectionné,
546              * on ajoute un ShapeFilter correspondant au type de figure (type)
547              * au drawing model
548              * Sinon on enlève du drawing modèle tout filtre correspondant
549              * au type de figure
550              */
551         }
552     }
553
554     /**
555      * Action réalisée pour ajouter ou retirer un filtre de type trait de figure
556      */
557     private class LineFilterAction extends AbstractAction // implements AboutHandler
558     {
559         /**
560          * Le type de trait de la figure
561          */
562         private LineType type;
563
564         /**
565          * Constructeur de l'action pour mettre en place ou enlever un filtre
566          * pour filtrer les types de figures
567          */
568         public LineFilterAction(LineType type)
569         {
570             this.type = type;
571             String name = type.toString();
572             putValue(LARGE_ICON_KEY, IconFactory.getIcon(name));
573             putValue(SMALL_ICON, IconFactory.getIcon(name + "_small"));
```



```
574             putValue(NAME, name);
575             putValue(SHORT_DESCRIPTION, "Set/unset " + name  + " filter");
576         }
577
578         /**
579          * Opérations réalisées par l'action
580          * @param event l'évènement déclenchant l'action. Peut provenir d'un
581          * bouton ou d'un item de menu
582          */
583         @Override
584         public void actionPerformed(ActionEvent event)
585         {
586             /*
587              * TODO Si l'AbstractButton de la source est sélectionné,
588              * on ajoute un LineFilter correspondant au type de trait de la
589              * figure (type) au drawing model
590              * Sinon on enlève du drawing modèle tout filtre correspondant
591              * au type de trait de la figure
592              */
593         }
594     }
595
596     /**
597      * Contrôleur d'évènement permettant de modifier le type de figures à
598      * dessiner.
599      * @note dépends de #drawingModel et #infoLabel qui doivent être non
600      * null avant instanciation
601      */
602     private class ShapeItemListener implements ItemListener
603     {
604         /**
605          * Constructeur valué du contrôleur.
606          * Initialise le type de dessin dans {@link EditorFrame#drawingModel}
607          * et crée le {@link AbstractCreationListener} correspondant.
608          * @param initialIndex l'index du type de forme sélectionné afin de
609          * mettre en place le bon creationListener dans le
610          * {@link EditorFrame#drawingPanel}.
611          */
612         public ShapeItemListener(FigureType type)
613         {
614             // TODO Mise en place du type de figure ds le drawingModel
615
616             /*
617              * TODO Création et Mise en place du creationListener adéquat
618              * dans le drawingPanel
619              */
620         }
621
622         @Override
623         public void itemStateChanged(ItemEvent e)
624         {
625             JComboBox<?> items = (JComboBox<?>) e.getSource();
626
627             /*
628              * TODO Récupération de l'index et vérification que le JCombobox
629              * a bien été changé puis
630              * Mise en place du type de figure ds le drawingModel
631              * Retrait du dernier creationListener du drawingPanel
632              * Création et Mise en place du creationListener adéquat
633              * dans le drawingPanel
634              */
635         }
636     }
637
638     /**
639      * Contrôleur d'évènements permettant de modifier la couleur du trait
640      * @note utilise #drawingModel qui doit être non null avant instanciation
641      */
642     private class ColorItemListener implements ItemListener
643     {
644         /**
645          * Ce à quoi s'applique la couleur choisie.
646          * Soit au rmplissage, soit au trait.
647          */
648         private PaintToType applyTo;
649
650         /**
651          * La dernière couleur choisie (pour le {@link JColorChooser})
652          */
653         private Color lastColor;
654
655         /**
```

```
656          * Le tableau des couleurs possibles
657          */
658         private Paint[] colors;
659 
660         /**
661          * L'index de la couleur spéciale à choisir avec un {@link JColorChooser}
662          */
663         private final int customColorIndex;
664 
665         /**
666          * L'index de la dernière couleur sélectionnée dans le combobox.
667          * Afin de pouvoir y revenir si jamais le {@link JColorChooser} est
668          * annulé.
669          */
670         private int lastSelectedIndex;
671 
672         /**
673          * la couleur choisie
674          */
675         private Paint paint;
676 
677         /**
678          * Constructeur du contrôleur d'évènements d'un combobox permettant
679          * de choisir la couleur de remplissage
680          * @param colors le tableau des couleurs possibles
681          * @param selectedIndex l'index de l'élément actuellement sélectionné
682          * @param customColorIndex l'index de la couleur spéciale parmis les
683          * colors à définir à l'aide d'un {@link JColorChooser}.
684          * @param applyTo Ce à quoi s'applique la couleur (le remplissage ou
685          * bien le trait)
686          */
687         public ColorItemListener(Paint[] colors,
688                                  int selectedIndex,
689                                  int customColorIndex,
690                                  PaintToType applyTo)
691         {
692             this.colors = colors;
693             lastSelectedIndex = selectedIndex;
694             this.customColorIndex = customColorIndex;
695             this.applyTo = applyTo;
696             lastColor = (Color) colors[selectedIndex];
697             paint = colors[selectedIndex];
698 
699             applyTo.applyPaintTo(paint, drawingModel);
700         }
701 
702         /**
703          * Actions à réaliser lorsque l'élément sélectionné du combox change
704          * @param e l'évènement de changement d'item du combobox
705          */
706         @Override
707         public void itemStateChanged(ItemEvent e)
708         {
709             JComboBox<?> combo = (JComboBox<?>) e.getSource();
710             int index = combo.getSelectedIndex();
711 
712             /*
713              * TODO Si l'index est correct (< colors.length)
714              *  Si l'état du combo est bien changé
715              *  Si l'item sélectionné correspond à customColorIndex
716              *  alors il faut ouvrir une Boite de dialoque de choix de couleur
717              *  avec la PaintFactory et si la couleur résultante est non null
718              *  l'appliquer au drawing model
719              *  sinon déterminer la couleur correspondant à l'item sélectionné
720              *  et l'appliquer au drawing model
721              */
722         }
723     }
724 
725     /**
726      * Contrôleur d'évènements permettant de modifier le type de trait (normal,
727      * pointillé, sans trait)
728      * @note utilise #drawingModel qui doit être non null avant instanciation
729      */
730     private class EdgeTypeListener implements ItemListener
731     {
732         /**
733          * Le type de trait à mettre en place
734          */
735         private LineType edgeType;
736 
737         public EdgeTypeListener(LineType type)
```



```
738         {
739             edgeType = type;
740             drawingModel.setEdgeType(edgeType);
741         }
742 
743         @Override
744         public void itemStateChanged(ItemEvent e)
745         {
746             JComboBox<?> items = (JComboBox<?>) e.getSource();
747             int index = items.getSelectedIndex();
748 
749             /*
750              * TODO Si l'état du combo est bien changé
751              * Récupérer le type de ligne correspondant à l'index sélectionné
752              * et l'applique au drawingModel
753              */
754         }
755     }
756 
757     /**
758      * Contrôleur d'évènement permettant de modifier la taille du trait
759      * en fonction des valeurs d'un {@link JSpinner}
760      */
761     private class EdgeWidthListener implements ChangeListener
762     {
763         /**
764          * Constructeur du contrôleur d'évènements contrôlant l'épaisseur du
765          * trait
766          * @param initialValue la valeur initiale de la largeur du trait à
767          * appliquer au dessin (EditorFrame#drawingModel)
768          */
769         public EdgeWidthListener(int initialValue)
770         {
771             drawingModel.setEdgeWidth(initialValue);
772         }
773 
774         /**
775          * Actions à réaliser lorsque la valeur du spinner change
776          * @param e l'évènement de changement de valeur du spinner
777          */
778         @Override
779         public void stateChanged(ChangeEvent e)
780         {
781             /*
782              * TODO récupérer le spinner d'après la source, puis
783              * son modèle et enfin mettre en place la valeur de l'épaisseur
784              * du trait dans le drawing model
785              */
786         }
787     }
788 }
```

```java
package widgets;

import java.awt.Color;
import java.awt.GridBagLayout;
import java.awt.Paint;
import java.text.DecimalFormat;
import java.util.HashMap;
import java.util.Map;

import javax.swing.ImageIcon;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.LineBorder;

import figures.Figure;
import figures.enums.FigureType;
import figures.enums.LineType;
import utils.IconFactory;

public class InfoPanel extends JPanel
{
    /**
     * Une chaine vide pour remplir les champs lorsque la souris n'est au dessus
     * d'aucune figure
     */
    private static final String emptyString = new String();

    /**
     * Une icône vide pour remplir les chanmps avec icône lorsque la souris
     * n'est au dessus d'aucune figure
     */
    private static final ImageIcon emptyIcon = IconFactory.getIcon("None");

    /**
     * Le formatteur à utiliser pour formater les coordonnés
     */
    private final static DecimalFormat coordFormat = new DecimalFormat("000");

    /**
     * Le label contenant le nom de la figure
     */
    private JLabel lblFigureName;

    /**
     * Le label contenant l'icône correspondant à la figure
     */
    private JLabel lblTypeicon;

    /**
     * La map contenant les différentes icônes des types de figures
     */
    private Map<FigureType, ImageIcon> figureIcons;

    /**
     * Le label contenant l'icône de la couleur de remplissage
     */
    private JLabel lblFillcolor;

    /**
     * Le label contenant l'icône de la couleur du contour
     */
    private JLabel lblEdgecolor;

    /**
     * Map contenant les icônes relatives aux différentes couleurs (de contour
     * ou de remplissage)
     */
    private Map<Paint, ImageIcon> paintIcons;

    /**
     * Le label contenant le type de contour
     */
    private JLabel lblStroketype;

    /**
     * Map contenant les icônes relatives au différents types de traits de
     * contour
     */
    private Map<LineType, ImageIcon> lineTypeIcons;

    /**
     * Le label contenant l'abcisse du point en haut à gauche de la figure
```



```java
     */
    private JLabel lblTlx;

    /**
     * Le label contenant l'ordonnée du point en haut à gauche de la figure
     */
    private JLabel lblTly;

    /**
     * Le label contenant l'abcisse du point en bas à droite de la figure
     */
    private JLabel lblBrx;

    /**
     * Le label contenant l'ordonnée du point en bas à droite de la figure
     */
    private JLabel lblBry;

    /**
     * Le label contenant la largeur de la figure
     */
    private JLabel lblDx;

    /**
     * Le label contenant la hauteur de la figure
     */
    private JLabel lblDy;

    /**
     * Le label contenant l'abcisse du barycentre de la figure
     */
    private JLabel lblCx;

    /**
     * Le label contenant l'ordonnée du barycentre de la figure
     */
    private JLabel lblCy;

    /**
     * Create the panel.
     */
    public InfoPanel()
    {
        // ----------------------------------------------------------------------
        // Initialisation des maps
        // ----------------------------------------------------------------------
        figureIcons = new HashMap<FigureType, ImageIcon>();
        // TODO Remplissage de figureIcons en utilisant l'IconFactory

        paintIcons = new HashMap<Paint, ImageIcon>();
        String[] colorStrings = {
            "Black",
            "Blue",
            "Cyan",
            "Green",
            "Magenta",
            "None",
            "Orange",
            "Others",
            "Red",
            "White",
            "Yellow"
        };

        /*
         * TODO Obtention des paints par la PaintFactory, puis
         * Obtention des icônes correspondant aux paints avec l'IconFactory
         * pour remplir paintIcons
         */

        lineTypeIcons = new HashMap<LineType, ImageIcon>();

        /*
         * TODO Remplissage de lineTypeIcons avec l'IconFactory pour chaque
         * type de lignes
         */

        // ----------------------------------------------------------------------
        // Création de l'UI
        // ----------------------------------------------------------------------
        setBorder(new LineBorder(new Color(0, 0, 0), 1, true));
        GridBagLayout gridBagLayout = new GridBagLayout();
```

```
165         gridBagLayout.columnWidths = new int[] {80, 60, 60};
166         gridBagLayout.rowHeights = new int[] {30, 32, 32, 32, 20, 20, 20, 20, 20};
167         gridBagLayout.columnWeights = new double[]{0.0, 0.0, 0.0};
168         gridBagLayout.rowWeights = new double[]{0.0, 0.0, 0.0, 0.0, 0.0};
169         setLayout(gridBagLayout);
170
171         // TODO Compléter ...
172     }
173
174     /**
175      * Mise à jour de tous les labels avec les informations de figure
176      * @param figure la figure dont il faut extraire les informations
177      */
178     public void updateLabels(Figure figure)
179     {
180         // TODO MAJ titre de la figure
181
182         // TODO MAJ Icône du type de figure
183
184         // TODO MAJ Icône de la couleur de remplissage
185
186         // TODO MAJ Icône de la couleur de trait
187
188         // TODO MAJ Icône du type de trait
189
190         /*
191          * TODO MAJ Données numériques de la figure en utilisant les méthodes
192          * de la classe Figure
193          *  - Top left corner x & y
194          *  - Bottom right corner ...
195          *  - Dimensions ...
196          *  - Center ...
197          */
198
199         /*
200          * TODO Formmattage des données numérique avec coordFormat pour mette
201          * à jour les différents label avec les nouvelles valeurs numériques
202          */
203
204     }
205
206     /**
207      * Effacement de tous les labels
208      */
209     public void resetLabels()
210     {
211         // TODO RAZ titre de la figure
212
213         // TODO RAZ Icône du type de figure
214
215         // TODO RAZ Icône de la couleur de remplissage
216
217         // TODO RAZ Icône de la couleur de trait
218
219         // TODO RAZ Icône du type de trait
220
221         // TODO RAZ Données numériques
222     }
223 }
```



```
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;
```