```
# 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-%H-%M-%S)
# 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/8/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/package-info.java \
$(SRC)/figures/Figure.java \
$(SRC)/figures/NGon.java \
$(SRC)/figures/Polygon.java \
$(SRC)/figures/Rectangle.java \
$(SRC)/figures/RoundedRectangle.java \
$(SRC)/figures/Circle.java \
$(SRC)/figures/Drawing.java \
$(SRC)/figures/Ellipse.java \
$(SRC)/figures/enums/package-info.java \
$(SRC)/figures/enums/FigureType.java \
$(SRC)/figures/enums/LineType.java \
$(SRC)/figures/enums/PaintToType.java \
$(SRC)/figures/listeners/package-info.java \
$(SRC)/figures/listeners/AbstractFigureListener.java \
$(SRC)/figures/listeners/SelectionFigureListener.java \
$(SRC)/figures/listeners/creation/package-info.java \
$(SRC)/figures/listeners/creation/AbstractCreationListener.java \
$(SRC)/figures/listeners/creation/NGonCreationListener.java \
$(SRC)/figures/listeners/creation/PolygonCreationListener.java \
$(SRC)/figures/listeners/creation/RectangularShapeCreationListener.java \
$(SRC)/figures/listeners/creation/RoundedRectangleCreationListener.java \
$(SRC)/figures/listeners/transform/package-info.java \
$(SRC)/figures/listeners/transform/AbstractTransformShapeListener.java \
$(SRC)/figures/listeners/transform/MoveShapeListener.java \
$(SRC)/figures/listeners/transform/RotateShapeListener.java \
$(SRC)/figures/listeners/transform/ScaleShapeListener.java \
$(SRC)/figures/treemodels/package-info.java \
$(SRC)/figures/treemodels/AbstractFigureTreeModel.java \
$(SRC)/figures/treemodels/FigureTreeModel.java \
$(SRC)/figures/treemodels/AbstractTypedFigureTreeModel.java \
$(SRC)/figures/treemodels/FigureTypeTreeModel.java \
$(SRC)/figures/treemodels/FillColorTreeModel.java \
$(SRC)/figures/treemodels/EdgeColorTreeModel.java \
$(SRC)/figures/treemodels/LineTypeTreeModel.java \
$(SRC)/filters/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/package-info.java \
$(SRC)/utils/PaintFactory.java \
```



```
$(SRC)/utils/StrokeFactory.java \
$(SRC)/utils/Vector2D.java \
$(SRC)/utils/CColor.java \
$(SRC)/history/package-info.java \
$(SRC)/history/HistoryManager.java \
$(SRC)/history/Memento.java \
$(SRC)/history/Originator.java \
$(SRC)/history/Prototype.java \
$(SRC)/widgets/EditorFrame.java \
$(SRC)/widgets/DrawingPanel.java \
$(SRC)/widgets/enums/OperationMode.java \
$(SRC)/widgets/enums/package-info.java \
$(SRC)/widgets/enums/TreeType.java \
$(SRC)/widgets/InfoPanel.java \
$(SRC)/widgets/JLabeledComboBox.java \
$(SRC)/widgets/package-info.java \
$(SRC)/widgets/TreesPanel.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/Clear.png \
$(SRC)/images/Clear_small.png \
$(SRC)/images/ClearFilter.png \
$(SRC)/images/ClearFilter_small.png \
$(SRC)/images/Creation.png \
$(SRC)/images/Creation_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/Details.png \
$(SRC)/images/Details_small.png \
$(SRC)/images/EdgeColor.png \
$(SRC)/images/EdgeColor_small.png \
$(SRC)/images/Edition.png \
$(SRC)/images/Edition_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/Move.png \
$(SRC)/images/Move_small.png \
$(SRC)/images/MoveDown.png \
$(SRC)/images/MoveDown_small.png \
$(SRC)/images/MoveUp.png \
$(SRC)/images/MoveUp_small.png \
$(SRC)/images/Ngon.png \
$(SRC)/images/Ngon_small.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/Redo.png \
$(SRC)/images/Redo_small.png \
"$(SRC)/images/Rounded Rectangle.png" \
"$(SRC)/images/Rounded Rectangle_small.png" \
$(SRC)/images/RoundedRectangle.png \
$(SRC)/images/RoundedRectangle_small.png \
$(SRC)/images/Solid.png \
$(SRC)/images/Solid_small.png \
$(SRC)/images/Star.png \
$(SRC)/images/Star_small.png \
$(SRC)/images/Style.png \
$(SRC)/images/Style_small.png \
$(SRC)/images/Tree.png \
$(SRC)/images/Tree_small.png \
$(SRC)/images/Undo.png \
$(SRC)/images/Undo_small.png \
$(SRC)/images/White.png \
$(SRC)/images/Yellow.png \
TP5.pdf

.PHONY : doc ps
```

```
181
182 # Les targets de compilation
183 # pour générer l'application
184 all : $(foreach name, $(MAIN), $(BIN)/$(name).class)
185
186 #règle de compilation générique
187 $(BIN)/%.class : $(SRC)/%.java
188     $(JAVAC) -sourcepath $(SRC) -classpath $(BIN):$(CLASSPATH) -d $(BIN) $(CFLAGS) $<
189
190 # Edition des sources $(EDITOR) doit être une variable d'environnement
191 edit :
192     $(EDITOR) $(SOURCES) Makefile &
193
194 # nettoyer le répertoire
195 clean :
196     find bin/ -type f -name "*.class" -exec rm -f {} \;
197     rm -rf *~ $(DOC)/* $(LISTDIR)/*
198
199 #realclean : clean
200 #    rm -f $(ARCHDIR)/*.$(ARCHFMT)
201
202 # générer le listing
203 $(LISTDIR) :
204     mkdir $(LISTDIR)
205
206 ps : $(LISTDIR)
207     $(A2PS) -2 --file-align=fill --line-numbers=1 --font-size=10 \
208     --chars-per-line=100 --tabsize=4 --pretty-print \
209     --highlight-level=heavy --prologue="gray" \
210     -o$(LISTDIR)/$(OUTPUT).ps Makefile $(SOURCES)
211
212 pdf : ps
213     $(PS2PDF) $(LISTDIR)/$(OUTPUT).ps $(LISTDIR)/$(OUTPUT).pdf
214
215 # générer le listing lisible pour Gérard
216 bigps :
217     $(A2PS) -1 --file-align=fill --line-numbers=1 --font-size=10 \
218     --chars-per-line=100 --tabsize=4 --pretty-print \
219     --highlight-level=heavy --prologue="gray" \
220     -o$(LISTDIR)/$(OUTPUT).ps Makefile $(SOURCES)
221
222 bigpdf : bigps
223     $(PS2PDF) $(LISTDIR)/$(OUTPUT).ps $(LISTDIR)/$(OUTPUT).pdf
224
225 # voir le listing
226 preview : ps
227     $(GHOSTVIEW) $(LISTDIR)/$(OUTPUT); rm -f $(LISTDIR)/$(OUTPUT) $(LISTDIR)/$(OUTPUT)~
228
229 # générer la doc avec javadoc
230 doc : $(SOURCES)
231     $(DOCP) -private -d $(DOC) -author -link $(LOCALLINK) $(SOURCES)
232 #    $(DOCP) -private -d $(DOC) -author -linkoffline $(WEBLINK) $(LOCALLINK) $(SOURCES)
233
234 # générer une archive de sauvegarde
235 $(ARCHDIR) :
236     mkdir $(ARCHDIR)
237
238 archive : pdf $(ARCHDIR)
239     $(ARCH) $(ARCHDIR)/$(ARCHIVE)-$(DATE).$(ARCHFMT) $(SOURCES) $(LISTDIR)/*.pdf $(OTHER) $(BIN) Mak
    efile $(FIGDIR)/*.pdf
240
241 # exécution des programmes de test
242 run : all
243     $(foreach name, $(MAIN), $(JAVA) -classpath $(BIN):$(CLASSPATH) $(name) $(JAVAOPTIONS) )
```



```
1  import java.awt.EventQueue;
2
3  import javax.swing.UIManager;
4  import javax.swing.UIManager.LookAndFeelInfo;
5  import javax.swing.UnsupportedLookAndFeelException;
6
7  import widgets.EditorFrame;
8
9  /**
10  * Programme principal lançant la fenêtre {@link EditorFrame}
11  * @author davidroussel
12  */
13 public class Editor
    {
15
16      /**
17       * Programme principal
18       * @param args arguments : le nom du look and feel à utiliser
19       */
20      public static void main(String[] args)
21      {
22          /*
23           * Mise ne place du look and feel du système, ou celui fourni en
24           * argument du programme
25           */
26          try
27          {
28              if (args.length == 0)
29              {
30                  UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
31              }
32              else
33              {
34                  String lookAndFeelName = args[0];
35                  LookAndFeelInfo[] lafis = UIManager.getInstalledLookAndFeels();
36                  for (LookAndFeelInfo lafi : lafis)
37                  {
38                      if (lafi.getName().toLowerCase().equals(lookAndFeelName.toLowerCase()))
39                      {
40                          UIManager.setLookAndFeel(lafi.getClassName());
41                          break;
42                      }
43                  }
44              }
45          }
46          catch (ClassNotFoundException e)
47          {
48              System.err.println("Look and feel could not be found");
49              e.printStackTrace();
50          }
51          catch (InstantiationException e)
52          {
53              System.err.println("new instance of the class couldn't be created");
54              e.printStackTrace();
55          }
56          catch (IllegalAccessException e)
57          {
58              System.err.println("Look and feel class or initializer isn't accessible");
59              e.printStackTrace();
60          }
61          catch (UnsupportedLookAndFeelException e)
62          {
63              System.err.println("isSupportedLookAndFeel() is false");
64              e.printStackTrace();
65          }
66          catch (ClassCastException e)
67          {
68              System.err.println("className does not identify a class that extends LookAndFeel");
69              e.printStackTrace();
70          }
71
72          // Mise en place spécifique à Mac OS X
73          String osName = System.getProperty("os.name");
74          if (osName.startsWith("Mac OS"))
75          {
76              macOSSettings();
77          }
78
79          /*
80           * Création de la fenêtre
81           */
82          final EditorFrame frame = new EditorFrame();
83
84          /*
85           * Insertion de la fenêtre dans la file des évènements GUI
86           */
87          EventQueue.invokeLater(new Runnable()
88          {
89              @Override
90              public void run()
```

```
91              {
92                  try
93                  {
94                      frame.pack();
95                      frame.setVisible(true);
96                  }
97                  catch (Exception e)
98                  {
99                      e.printStackTrace();
100                 }
101             }
102         });
103
104     }
105
106     /**
107      * Mise en place des options spécifiques à MacOS.
108      * A virer si votre système n'est pas MacOS car com.apple.... risque
109      * de ne pas exister
110      */
111     private static void macOSSettings()
112     {
113         // Remettre les menus au bon endroit (dans la barre en haut)
114         System.setProperty("apple.laf.useScreenMenuBar", "true");
115
116 //      ImageIcon imageIcon = new ImageIcon(
117 //              Editor.class.getResource("/images/Logo.png"));
118 //      if (imageIcon.getImageLoadStatus() == MediaTracker.COMPLETE)
119 //      {
120             // Titre de l'application
121             System.setProperty(
122                     "com.apple.mrj.application.apple.menu.about.name",
123                     "Figure Editor");
124 //          // Chargement d'une icône pour le dock
125 //          com.apple.eawt.Application.getApplication().setDockIconImage(
126 //                  imageIcon.getImage());
127 //      }
128     }
129 }
130
```



```
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  @SuppressWarnings("serial")
52  public class ShapesDemo2D extends JApplet
53  {
54      protected final static int maxCharHeight = 15;
55      protected final static int minFontSize = 6;
56
57      protected final static Color bg = Color.white;
58      protected final static Color fg = Color.black;
59      protected final static Color red = Color.red;
60      protected final static Color white = Color.white;
61
62      protected final static BasicStroke stroke = new BasicStroke(2.0f);
63      protected final static BasicStroke wideStroke = new BasicStroke(8.0f,
64              BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
65
66      protected final static float lastWidth = 20.0f;
67      protected final static float dash1[] = { 2*lastWidth };
68      protected final static BasicStroke dashed = new BasicStroke(1.0f,
69              BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, 10.0f, dash1, 0.0f);
70      protected final static BasicStroke fatDashed = new BasicStroke(lastWidth,
71              BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND, lastWidth, dash1, 0.0f);
72      protected Dimension totalSize;
73      protected FontMetrics fontMetrics;
74
75      @Override
76      public void init()
77      {
78          // Initialize drawing colors
79          setBackground(bg);
80          setForeground(fg);
81      }
82
83      FontMetrics pickFont(Graphics2D g2, String longString, int xSpace)
84      {
85          boolean fontFits = false;
86          Font font = g2.getFont();
87          FontMetrics fontMetrics = g2.getFontMetrics();
88          int size = font.getSize();
89          String name = font.getName();
90          int style = font.getStyle();
```

```
91
92          while (¬fontFits)
93          {
94              if ((fontMetrics.getHeight() ≤ maxCharHeight)
95                      ∧ (fontMetrics.stringWidth(longString) ≤ xSpace))
96              {
97                  fontFits = true;
98              }
99              else
100             {
101                 if (size ≤ minFontSize)
102                 {
103                     fontFits = true;
104                 }
105                 else
106                 {
107                     g2.setFont(font = new Font(name, style, --size));
108                     fontMetrics = g2.getFontMetrics();
109                 }
110             }
111         }
112
113         return fontMetrics;
114     }
115
116     @Override
117     public void paint(Graphics g)
118     {
119         Graphics2D g2 = (Graphics2D) g;
120         g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
121                 RenderingHints.VALUE_ANTIALIAS_ON);
122         Dimension d = getSize();
123         int gridWidth = d.width / 6;
124         int gridHeight = d.height / 2;
125
126         fontMetrics = pickFont(g2, "Filled and Stroked GeneralPath", gridWidth);
127
128         Color fg3D = Color.lightGray;
129
130         // on commence par effacer le fond
131         g2.setColor(getBackground());
132         g2.fillRect(0, 0, d.width, d.height);
133
134         g2.setPaint(fg3D);
135         g2.draw3DRect(0, 0, d.width - 1, d.height - 1, true);
136         g2.draw3DRect(3, 3, d.width - 7, d.height - 7, false);
137         g2.setPaint(fg);
138
139         int x = 5;
140         int y = 7;
141         int rectWidth = gridWidth - (2 * x);
142         int stringY = gridHeight - 3 - fontMetrics.getDescent();
143         int rectHeight = stringY - fontMetrics.getMaxAscent() - y - 2;
144
145         // draw Line2D.Double
146         g2.draw(new Line2D.Double(x, (y + rectHeight) - 1, x + rectWidth, y));
147         g2.drawString("Line2D", x, stringY);
148         x += gridWidth;
149
150         // draw Rectangle2D.Double
151         g2.setStroke(stroke);
152         g2.draw(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
153         g2.drawString("Rectangle2D", x, stringY);
154         x += gridWidth;
155
156         // draw RoundRectangle2D.Double
157         g2.setStroke(dashed);
158         g2.draw(new RoundRectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10));
159         g2.drawString("RoundRectangle2D", x, stringY);
160         x += gridWidth;
161
162         // draw Arc2D.Double
163         g2.setStroke(wideStroke);
164         g2.draw(new Arc2D.Double(x, y, rectWidth, rectHeight, 90, 135,
165                 Arc2D.OPEN));
166         g2.drawString("Arc2D", x, stringY);
167         x += gridWidth;
168
169         // draw Ellipse2D.Double
170         g2.setStroke(stroke);
171         g2.draw(new Ellipse2D.Double(x, y, rectWidth, rectHeight));
172         g2.drawString("Ellipse2D", x, stringY);
173         x += gridWidth;
174
175         // draw GeneralPath (polygon)
176         int x1Points[] = { x, x + rectWidth, x, x + rectWidth };
177         int y1Points[] = { y, y + rectHeight, y + rectHeight, y };
178         GeneralPath polygon = new GeneralPath(Path2D.WIND_EVEN_ODD,
179                 x1Points.length);
180         polygon.moveTo(x1Points[0], y1Points[0]);
```



```
181         for (int index = 1; index < x1Points.length; index++)
182         {
183             polygon.lineTo(x1Points[index], y1Points[index]);
184         }
185
186         polygon.closePath();
187
188         g2.draw(polygon);
189         g2.drawString("GeneralPath", x, stringY);
190
191         // NEW ROW
192         x = 5;
193         y += gridHeight;
194         stringY += gridHeight;
195
196         // draw GeneralPath (polyline)
197
198         int x2Points[] = { x, x + rectWidth, x, x + rectWidth };
199         int y2Points[] = { y, y + rectHeight, y + rectHeight, y };
200         GeneralPath polyline = new GeneralPath(Path2D.WIND_EVEN_ODD,
201                 x2Points.length);
202         polyline.moveTo(x2Points[0], y2Points[0]);
203         for (int index = 1; index < x2Points.length; index++)
204         {
205             polyline.lineTo(x2Points[index], y2Points[index]);
206         }
207
208         g2.draw(polyline);
209         g2.drawString("GeneralPath (open)", x, stringY);
210         x += gridWidth;
211
212         // fill Rectangle2D.Double (red)
213         g2.setPaint(red);
214         g2.fill(new Rectangle2D.Double(x, y, rectWidth, rectHeight));
215         g2.setPaint(fg);
216         g2.drawString("Filled Rectangle2D", x, stringY);
217         x += gridWidth;
218
219         // fill RoundRectangle2D.Double
220         GradientPaint redtowhite = new GradientPaint(x, y, red, x + rectWidth,
221                 y, white);
222         g2.setPaint(redtowhite);
223         g2.fill(new RoundRectangle2D.Double(x, y, rectWidth, rectHeight, 10, 10));
224         g2.setPaint(fg);
225         g2.drawString("Filled RoundRectangle2D", x, stringY);
226         x += gridWidth;
227
228         // fill Arc2D
229         g2.setPaint(red);
230         g2.fill(new Arc2D.Double(x, y, rectWidth, rectHeight, 90, 135,
231                 Arc2D.OPEN));
232         g2.setPaint(fg);
233         g2.drawString("Filled Arc2D", x, stringY);
234         x += gridWidth;
235
236         // fill Ellipse2D.Double
237         redtowhite = new GradientPaint(x, y, red, x + rectWidth, y, white);
238         g2.setPaint(redtowhite);
239         g2.fill(new Ellipse2D.Double(x, y, rectWidth, rectHeight));
240         g2.setPaint(fg);
241         g2.drawString("Filled Ellipse2D", x, stringY);
242         x += gridWidth;
243
244         // fill and stroke GeneralPath
245         int x3Points[] = { x, x + rectWidth, x, x + rectWidth };
246         int y3Points[] = { y, y + rectHeight, y + rectHeight, y };
247         GeneralPath filledPolygon = new GeneralPath(Path2D.WIND_EVEN_ODD,
248                 x3Points.length);
249         filledPolygon.moveTo(x3Points[0], y3Points[0]);
250         for (int index = 1; index < x3Points.length; index++)
251         {
252             filledPolygon.lineTo(x3Points[index], y3Points[index]);
253         }
254         filledPolygon.closePath();
255
256         g2.setPaint(red);
257         g2.fill(filledPolygon);
258
259         g2.setStroke(fatDashed);
260         g2.setPaint(fg);
261         g2.draw(filledPolygon);
262
263         g2.drawString("Filled and Stroked GeneralPath", x, stringY);
264     }
265
266     public static void main(String s[])
267     {
268         JFrame f = new JFrame("ShapesDemo2D");
269         f.addWindowListener(new WindowAdapter()
270         {
```

```
271             @Override
272             public void windowClosing(WindowEvent e)
273             {
274                 System.exit(0);
275             }
276         });
277         JApplet applet = new ShapesDemo2D();
278         f.getContentPane().add("Center", applet);
279         applet.init();
280         f.pack();
281         f.setSize(new Dimension(550, 100));
282         f.setVisible(true);
283     }
284
285 }
```



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

```java
package figures;

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

import figures.enums.FigureType;
import figures.enums.LineType;
import history.Prototype;
import utils.CColor;
import utils.PaintFactory;
import utils.StrokeFactory;

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

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

    /**
     * Couleur du bord de la sélection
     */
    protected static final Paint selectedEdge =
        PaintFactory.getPaint(Color.LIGHT_GRAY);

    /**
     * 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;

    /**
     * Caractéristique de la bordure des figures sélectionnées
     */
    protected static final BasicStroke selectedStroke =
        StrokeFactory.getStroke(LineType.DASHED, 2.0f);

    /**
     * La translation à appliquer à cet objet
     * @note sert à déplacer cet objet, pour ce faire il faut
     * avant de dessiner cet objet appliquer cette transformation ainsi que
     * sa rotation et son facteur d'échelle puis les retirer après le dessin.
     */
    protected AffineTransform translation;

    /**
     * La rotation à appliquer à cet objet
     * @note sert à tourner cet objet, pour ce faire il faut
     * avant de dessiner cet objet appliquer cette transformation ainsi que
     * sa translation et son facteur d'échelle puis les retirer après le dessin.
     */
    protected AffineTransform rotation;

    /**
     * Le facteur d'échelle à applique à cet objet
     * @note sert à changer l'échelle cet objet, pour ce faire il faut
     * avant de dessiner cet objet appliquer cette transformation ainsi que
     * sa translation et sa rotation puis les retirer après le dessin.
     */
    protected AffineTransform scale;

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

    /**
     * Indique si la figure fait partie des figrues sélectionnées
```



```java
     */
    protected boolean selected;

    /**
     * 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;
        translation = new AffineTransform();
        translation.setToIdentity();
        rotation = new AffineTransform();
        rotation.setToIdentity();
        scale = new AffineTransform();
        scale.setToIdentity();
        selected = false;
    }

    /**
     * Constructeur de copie assurant une copie distincte de la figure
     * @param f la figure à copier
     */
    protected Figure(Figure f)
    {
        shape = null; // Shapes must be copied in subclasses
        edge = PaintFactory.getPaint(f.edge);
        fill = PaintFactory.getPaint(f.fill);
        stroke = StrokeFactory.getStroke(f.stroke);
        translation = new AffineTransform(f.translation);
        rotation = new AffineTransform(f.rotation);
        scale = new AffineTransform(f.scale);
        instanceNumber = f.instanceNumber;
        selected = f.selected;
    }

    /**
     * Création d'une copie distincte de la figure
     */
    @Override
    public abstract Figure clone();

    /**
     * Comparaison de deux figures
     * @param Object o l'objet à comparer
     * @return true si obj est une figure de même type et que son contenu est
     * identique
     */
    @Override
    public boolean equals(Object o)
    {
        if (o == null)
        {
            return false;
        }

        if (o == this)
        {
            return true;
        }

        if (getClass() == o.getClass())
        {
            Figure f = (Figure) o;

            if (getType().equals(f.getType()))
            {
                if (instanceNumber == f.instanceNumber)
                {
                    // boolean edgeTest = (edge == null ? f.edge == null : edge.equals(f.edge));
                    /*
                     * Les egde sont fournies par une PaintFactory donc elles sont uniques
                     */
                    boolean edgeTest = (edge == f.edge);
                    if (edgeTest)
                    {
                        // boolean fillTest = (fill == null ? f.fill == null : fill.equals(f.fill));
                        /*
                         * Les fill sont fournies par une PaintFactory donc elles sont uniques
                         */
```

```
                        boolean fillTest = (fill == f.fill);
                        if (fillTest)
                        {
                            // boolean strokeTest = (stroke == null ?
                            //                       f.stroke == null :
                            //                      stroke.equals(f.stroke));
                            /*
                             * Les stroke sont fournies par une StrokeFactory donc elles sont unique
s
                             */
                            boolean strokeTest = (stroke == f.stroke);
                            if (strokeTest)
                            {
                                if (translation.equals(f.translation))
                                {
                                    if (rotation.equals(f.rotation))
                                    {
                                        if (scale.equals(f.scale))
                                        {
                                            if (getCenter()
                                                .equals(f.getCenter()))
                                            {
                                                if (getBounds2D()
                                                    .equals(f.getBounds2D()))
                                                {
                                                    return true;
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }

        return false;
    }

    /**
     * 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)
    {
        // Get the current transform
         AffineTransform savedT = g2D.getTransform();

         // Perform transformations
         g2D.transform(getTransform());

         // Render
        if (fill ≠ null)
        {
            g2D.setPaint(fill);
            g2D.fill(shape);
        }
        if ((edge ≠ null) ∧ (stroke ≠ null))
        {
            g2D.setStroke(stroke);
            g2D.setPaint(edge);
            g2D.draw(shape);
        }

        // Restore original transform
        g2D.setTransform(savedT);
    }

    /**
     * Dessin de la sélection de la figure (son soulignement) dans un contexte
     * graphique fournit par le système.
     * @note le dessin de la sélection doit être séparé du dessin des figures
     * de manière à ce que les sélection apparaissent par dessus les figures
     * @param g2D le contexte graphique
     */
    public final void drawSelection(Graphics2D g2D)
```



```
    {
        if (selected)
        {
            g2D.setPaint(selectedEdge);
            g2D.setStroke(selectedStroke);
            g2D.draw(getBounds2D()); // getBounds uses current transform
        }
    }

    /**
     * Normalise une figure de manière à exprimer tous ses points par rapport
     * à son centre, puis transfère la position réelle du centre dans l'attribut
     * {@link #translation}
     */
    public abstract void normalize();

    /**
     * Accesseur en lecture de la translation courante
     * @return la translation courante
     */
    public AffineTransform getTranslation()
    {
        return translation;
    }

    /**
     * Accesseur en lecture de la rotation courante
     * @return la rotation courante
     */
    public AffineTransform getRotation()
    {
        return rotation;
    }

    /**
     * Accesseur en lecture de l'échelle courante
     * @return l'échelle courante
     */
    public AffineTransform getScale()
    {
        return scale;
    }

    /**
     * Produit la transformation complète de cet objet
     * (facteur d'échelle)*(rotation)*(translation)
     * @return la transformation combinant le facteur d'échelle, la rotation et
     * la translation de cette figure.
     */
    public AffineTransform getTransform()
    {
        AffineTransform transform = (AffineTransform) translation.clone();
        transform.concatenate(scale);
        transform.concatenate(rotation);

        return transform;
    }

    /**
     * Mise en place d'une translation
     * @param translation la translation à mettre en place
     */
    public void setTranslation(AffineTransform translation)
    {
        this.translation = translation;
    }

    /**
     * Déplace la figure de dx et dy
     * @param dx la variation d'abcisse de la figure
     * @param dy la variation d'ordonnée de la figure
     */
    public void translate(double dx, double dy)
    {
        translation.translate(dx, dy);
    }

    /**
     * Mise en place d'une rotation
     * @param rotation la rotation à mettre en place
     */
    public void setRotation(AffineTransform rotation)
    {
        this.rotation = rotation;
    }

    /**
     * Fait tourner la figure d'un certain angle autour de son barycentre
     * @param deltaAngle l'angle de rotation de la figure
     */
```

```
360     public void rotate(double deltaAngle)
361     {
362         rotation.rotate(deltaAngle);
363     }
364
365     /**
366      * Mise en place d'un facteur d'échelle
367      * @param scale le facteur d'échelle à mettre en place
368      */
369     public void setScale(AffineTransform scale)
370     {
371         this.scale = scale;
372     }
373
374     /**
375      * Change l'échelle de la figure
376      * @param deltaScale le facteur d'échelle àç appliquer à la figure
377      */
378     public void scale(double deltaScale)
379     {
380         scale.scale(deltaScale, deltaScale);
381     }
382
383     /**
384      * Obtention du rectangle englobant de la figure.
385      * Obtenu grâce au {@link Shape#getBounds2D()}
386      * @return le rectangle englobant de la figure
387      */
388     public Rectangle2D getBounds2D()
389     {
390         /*
391          * Attention, il faut appliquer la transformation affine courante
392          * au Rectangle2D résultant de l'appel à shape.getBounds2D();
393          */
394         Rectangle2D bounds = shape.getBounds2D();
395         double minX = bounds.getMinX();
396         double minY = bounds.getMinY();
397         double maxX = bounds.getMaxX();
398         double maxY = bounds.getMaxY();
399         Point2D[] corners = new Point2D[] {
400             new Point2D.Double(minX, minY),
401             new Point2D.Double(maxX, minY),
402             new Point2D.Double(maxX, maxY),
403             new Point2D.Double(minX, maxY)
404         };
405         Point2D[] tCorners = new Point2D[4];
406         for (int i = 0; i < 4; i++)
407         {
408             tCorners[i] = new Point2D.Double();
409         }
410
411         getTransform().transform(corners, 0, tCorners, 0, corners.length);
412
413         double x = tCorners[0].getX();
414         double y = tCorners[0].getY();
415         double w = 0.0;
416         double h = 0.0;
417
418         for (int i = 0; i < 4; i++)
419         {
420             double tx = tCorners[i].getX();
421             x = (x < tx ? x : tx);
422
423             double ty = tCorners[i].getY();
424             y = (y < ty ? y : ty);
425         }
426
427         for (int i = 0; i < 4; i++)
428         {
429             double tw = tCorners[i].getX() - x;
430             w = (tw > w ? tw : w);
431
432             double th = tCorners[i].getY() - y;
433             h = (th > h ? th : h);
434         }
435
436         bounds.setFrame(x, y, w, h);
437
438         return bounds;
439     }
440
441     /**
442      * Obtention du barycentre de la figure.
443      * @return le point correspondant au barycentre de la figure
444      */
445     public abstract Point2D getCenter();
446
447     /**
448      * Teste si le point p est contenu dans cette figure.
449      * Utilise {@link Shape#contains(Point2D)}
```



```
450      * @param p le point dont on veut tester s'il est contenu dans la figure
451      * @return true si le point p est contenu dans la figure, false sinon
452      */
453     public boolean contains(Point2D p)
454     {
455         /*
456          * TODO Ce point p doit subir la transformation inverse
457          * de celle subie par la figure pour déterminer si le point p fait
458          * partie de la figure
459          */
460         return false;
461     }
462
463     /**
464      * Accesseur du type de figure selon {@link FigureType}
465      * @return le type de figure
466      */
467     public abstract FigureType getType();
468
469     /**
470      * Accesseur en lecture du {@link Paint} du contour
471      * @return le {@link Paint} du contour
472      */
473     public Paint getEdgePaint()
474     {
475         return edge;
476     }
477
478     /**
479      * Accesseur en lecture de la couleur comparable du contour
480      * @return la couleur comparable du contour
481      */
482     public CColor getEdgeCColor()
483     {
484         if (edge ≠ null)
485         {
486             if (edge instanceof Color)
487             {
488                 return new CColor((Color)edge);
489             }
490         }
491
492         return CColor.NoColor;
493     }
494
495     /**
496      * Mutateur du {@link Paint} du contour
497      * @param edge le nouveau {@link Paint} à mettre dans {@link #edge}
498      */
499     public void setEdgePaint(Paint edge)
500     {
501         if (edge ≠ null)
502         {
503             this.edge = edge;
504         }
505         else
506         {
507             System.err.println(getClass().getSimpleName() + "::setEdgePaint : null paint");
508         }
509     }
510
511     /**
512      * Accesseur en lecture du {@link Paint} du remplissage
513      * @return le {@link Paint} du remplissage
514      */
515     public Paint getFillPaint()
516     {
517         return fill;
518     }
519
520     /**
521      * Accesseur en lecture de la couleur comparable de remplissage
522      * @return la couleur comparable du remplissage
523      */
524     public CColor getFillCColor()
525     {
526         if (fill ≠ null)
527         {
528             if (fill instanceof Color)
529             {
530                 return new CColor((Color)fill);
531             }
532         }
533
534         return CColor.NoColor;
535     }
536
537     /**
538      * Mutateur du {@link Paint} du contour
539      * @param fill le nouveau {@link Paint} à mettre dans {@link #fill}
```

```
540      */
541     public void setFillPaint(Paint fill)
542     {
543         if (fill ≠ null)
544         {
545             this.fill = fill;
546         }
547         else
548         {
549             System.err.println(getClass().getSimpleName() + "::setFillPaint : null paint");
550         }
551     }
552
553     /**
554      * Accesseur en lecture du type de ligne ({@link LineType}) en fonction
555      * du {@link #stroke}
556      * @return le type de ligne actuel d'après le {@link #stroke}.
557      * @see LineType#fromStroke(BasicStroke)
558      */
559     public LineType getLineType()
560     {
561         return LineType.fromStroke(stroke);
562     }
563
564     /**
565      * Accesseur en lecture du {@link BasicStroke} du contour
566      * @return le {@link BasicStroke} du contour
567      */
568     public BasicStroke getStroke()
569     {
570         return stroke;
571     }
572
573     /**
574      * Mutateur du {@link BasicStroke} du contour
575      * @param stroke le nouveau {@link BasicStroke} à mettre dans {@link #stroke}
576      */
577     public void setStroke(BasicStroke stroke)
578     {
579         if (stroke ≠ null)
580         {
581             this.stroke = stroke;
582         }
583         else
584         {
585             System.err.println(getClass().getSimpleName() + "::setStroke : null stroke");
586         }
587     }
588
589     /**
590      * Accesseur de la propriété {@link #selected}
591      * @return la valeur de {@link #selected}
592      */
593     public boolean isSelected()
594     {
595         return selected;
596     }
597
598     /**
599      * Mutateur de la propriété {@link #selected}
600      * @param selected la nouvelle valeur de selected
601      */
602     public void setSelected(boolean selected)
603     {
604         this.selected = selected;
605     }
606
607     @Override
608     public String toString()
609     {
610         return getClass().getSimpleName() + " " + String.valueOf(instanceNumber);
611     }
612 }
```



```
1  package figures;
2
3  import java.awt.BasicStroke;
4  import java.awt.Paint;
5  import java.awt.geom.Point2D;
6  import java.awt.geom.Rectangle2D;
7  import java.awt.geom.RectangularShape;
8
9  import figures.enums.FigureType;
10
11 /**
12  * Classe de Rectangle pour les {@link Figure}
13  *
14  * @author davidroussel
15  */
16 public class Rectangle extends Figure
17 {
18     /**
19      * Le compteur d'instance des cercles.
20      * Utilisé pour donner un numéro d'instance après l'avoir incrémenté
21      */
22     private static int counter = 0;
23
24     /**
25      * Création d'un rectangle avec les points en haut à gauche et en bas à
26      * droite
27      *
28      * @param stroke le type de trait
29      * @param edge la couleur du trait
30      * @param fill la couleur de remplissage
31      * @param topLeft le point en haut à gauche
32      * @param bottomRight le point en bas à droite
33      */
34     public Rectangle(BasicStroke stroke, Paint edge, Paint fill, Point2D topLeft,
35             Point2D bottomRight)
36     {
37         super(stroke, edge, fill);
38         instanceNumber = ++counter;
39         double x = topLeft.getX();
40         double y = topLeft.getY();
41         double w = (bottomRight.getX() - x);
42         double h = (bottomRight.getY() - y);
43
44         shape = new Rectangle2D.Double(x, y, w, h);
45
46         // System.out.println("Rectangle created");
47     }
48
49     /**
50      * Constructeur de copie assurant une copie distincte du rectangle
51      * @param rect le rectangle à copier
52      */
53     public Rectangle(Rectangle rect)
54     {
55         super(rect);
56         if (rect.getClass() ≡ Rectangle.class)
57         {
58             Rectangle2D oldRectangle = (Rectangle2D) rect.shape;
59             shape = new Rectangle2D.Double(oldRectangle.getMinX(),
60                                            oldRectangle.getMinY(),
61                                            oldRectangle.getWidth(),
62                                            oldRectangle.getHeight());
63         }
64         else
65         {
66             System.out.println("Calling Rectangle(Rectangle) from another class");
67         }
68     }
69
70     /**
71      * Création d'une copie distincte de la figure
72      * @see figures.Figure#clone()
73      */
74     @Override
75     public Figure clone()
76     {
77         return new Rectangle(this);
78     }
79
80     /**
81      * Comparaison de deux figures
82      * @param Object o l'objet à comparer
83      * @return true si obj est une figure de même type et que son contenu est
84      * identique
85      */
86     @Override
87     public boolean equals(Object o)
88     {
89         if (super.equals(o))
90         {
```

```
91              Rectangle r = (Rectangle) o;
92              RectangularShape r1 = (RectangularShape) shape;
93              RectangularShape r2 = (RectangularShape) r.shape;
94
95              return ((r1.getX() ≡ r2.getX()) ∧
96                      (r1.getY() ≡ r2.getY()) ∧
97                      (r1.getWidth() ≡ r2.getWidth()) ∧
98                      (r1.getHeight() ≡ r2.getHeight()));
99          }
100
101         return false;
102     }
103
104     /**
105      * Création d'un rectangle sans points (utilisé dans les classes filles
106      * pour initialiser seulement les couleur et le style de trait sans
107      * initialiser {@link #shape}.
108      *
109      * @param stroke le type de trait
110      * @param edge la couleur du trait
111      * @param fill la couleur de remplissage
112      */
113     protected Rectangle(BasicStroke stroke, Paint edge, Paint fill)
114     {
115         super(stroke, edge, fill);
116
117         shape = null;
118     }
119
120     /**
121      * Déplacement du point en bas à droite du rectangle à la position
122      * du point p
123      *
124      * @param p la nouvelle position du dernier point
125      * @see figures.Figure#setLastPoint(Point2D)
126      */
127     @Override
128     public void setLastPoint(Point2D p)
129     {
130         if (shape ≠ null)
131         {
132             Rectangle2D.Double rect = (Rectangle2D.Double) shape;
133             double newWidth = p.getX() - rect.x;
134             double newHeight = p.getY() - rect.y;
135             rect.width = newWidth;
136             rect.height = newHeight;
137         }
138         else
139         {
140             System.err.println(getClass().getSimpleName() + "::setLastPoint : null shape");
141         }
142     }
143
144     /**
145      * Obtention du barycentre de la figure.
146      * @return le point correspondant au barycentre de la figure
147      */
148     @Override
149     public Point2D getCenter()
150     {
151         RectangularShape rect = (RectangularShape) shape;
152
153         Point2D center = new Point2D.Double(rect.getCenterX(), rect.getCenterY());
154         Point2D tCenter = new Point2D.Double();
155         getTransform().transform(center, tCenter);
156
157         return tCenter;
158     }
159
160     /**
161      * Normalise une figure de manière à exprimer tous ses points par rapport
162      * à son centre, puis transfère la position réelle du centre dans l'attribut
163      * {@link #translation}
164      */
165     @Override
166     public void normalize()
167     {
168         Point2D center = getCenter();
169         double cx = center.getX();
170         double cy = center.getY();
171         RectangularShape rectangle = (RectangularShape) shape;
172         translation.translate(cx, cy);
173         rectangle.setFrame(rectangle.getX() - cx,
174                            rectangle.getY() - cy,
175                            rectangle.getWidth(),
176                            rectangle.getHeight());
177     }
178
179     /**
180      * Accesseur du type de figure selon {@link FigureType}
```



```
181      * @return le type de figure
182      */
183     @Override
184     public FigureType getType()
185     {
186         return FigureType.RECTANGLE;
187     }
188 }
```

```java
package figures;

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

import figures.enums.FigureType;
import figures.enums.LineType;
import filters.FigureFilters;
import history.Memento;
import history.Originator;
import utils.StrokeFactory;

/**
 * Classe contenant l'ensemble des figures à dessiner (LE MODELE)
 * @author davidroussel
 */
public class Drawing extends Observable implements Originator<Figure>
{
    /**
     * Liste des figures à dessiner (protected pour que les classes du même
     * package puissent y accéder)
     */
    protected Vector<Figure> figures;

    /**
     * Liste triée des indices (uniques) des figures sélectionnées
     */
    protected SortedSet<Integer> selectionIndex;

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

    /**
     * Le type de figure à créer (pour la prochaine figure)
     */
    private FigureType type;

    /**
     * La couleur de remplissage courante (pour la prochaine figure)
     */
    private Paint fillPaint;

    /**
     * La couleur de trait courante (pour la prochaine figure)
     */
    private Paint edgePaint;

    /**
     * La largeur de trait courante (pour la prochaine figure)
     */
    private float edgeWidth;

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

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

    /**
     * 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
```



```java
     * 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<Paint> fillColorFilter; // TODO décommenter lorsque prêt

    /**
     * Filtre à appliquer au flux des figures pour sélectionner les figures
     * ayant une couleur particulière de trait
     */
    // private EdgeColorFilter<Paint> edgeColorFilter; // TODO décommenter lorsque prêt

    /**
     * Filtres à applique 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>();
        selectionIndex = new TreeSet<Integer>(Integer::compare);
        shapeFilters = new FigureFilters<FigureType>();

        // fillColorFilter = null; // TODO décommenter lorsque prêt
        // edgeColorFilter = null; // TODO décommenter lorsque prêt
        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();
        figures = null;
        selectionIndex.clear();
        selectionIndex = null;
        fillPaint = null;
        edgePaint = null;
        edgeType = null;
        stroke = null;
        shapeFilters.clear();
        shapeFilters = null;
        // fillColorFilter = null; // TODO décommenter lorsque prêt
        // edgeColorFilter = null; // TODO décommenter lorsque prêt
        lineFilters.clear();
        lineFilters = null;
    }

    /**
     * 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(); // pour que les observateurs soient mis à jour
    }

    /**
     * Mise en place d'un nouveau type de figure à générer
     * @param type le nouveau type de figure
     */
    public void setFigureType(FigureType type)
    {
        this.type = type;
    }

    /**
     * Accesseur de la couleur de remplissage courante des figures
```

```
181      * @return la couleur de remplissage courante des figures
182      */
183     public Paint getFillpaint()
184     {
185         return fillPaint;
186     }
187 
188     /**
189      * Mise en place d'une nouvelle couleur de remplissage
190      * @param fillPaint la nouvelle couleur de remplissage
191      */
192     public void setFillPaint(Paint fillPaint)
193     {
194         this.fillPaint = fillPaint;
195     }
196 
197     /**
198      * Accesseur de la couleur de trait courante des figures
199      * @return la couleur de remplissage courante des figures
200      */
201     public Paint getEdgePaint()
202     {
203         return edgePaint;
204     }
205 
206     /**
207      * Mise en place d'une nouvelle couleur de trait
208      * @param edgePaint la nouvelle couleur de trait
209      */
210     public void setEdgePaint(Paint edgePaint)
211     {
212         this.edgePaint = edgePaint;
213     }
214 
215     /**
216      * Accesseur du trait courant des figures
217      * @return le trait courant des figures
218      */
219     public BasicStroke getStroke()
220     {
221         return stroke;
222     }
223 
224     /**
225      * Mise en place d'un nouvelle épaisseur de trait
226      * @param width la nouvelle épaisseur de trait
227      */
228     public void setEdgeWidth(float width)
229     {
230         edgeWidth = width;
231         /*
232          * TODO Il faut regénérer le stroke
233          */
234     }
235 
236     /**
237      * Mise en place d'un nouvel état de ligne pointillée
238      * @param type le nouveau type de ligne
239      */
240     public void setEdgeType(LineType type)
241     {
242         edgeType = type;
243         /*
244          * TODO Il faut regénérer le stroke
245          */
246 
247     }
248 
249     /**
250      * Initialisation d'une figure de type {@link #type} au point p et ajout de
251      * cette figure à la liste des {@link #figures}
252      * @param p le point où initialiser la figure
253      * @return la nouvelle figure créée à x et y avec les paramètres courants
254      */
255     public Figure initiateFigure(Point2D p)
256     {
257         /*
258          * TODO Maintenant que l'on s'apprête effectivement à créer une figure on
259          * ajoute/obtient les Paints et le Stroke des factories
260          */
261 
262         /*
263          * TODO Obtention de la figure correspondant au type de figure choisi grâce à
264          * type.getFigure(...)
265          */
266         Figure newFigure = null; // TODO remplacer par type.getFigure(...)
267 
268         /*
269          * TODO Ajout de la figure à #figures
270          */
```



```
271 
272         /* TODO Notification des observers */
273 
274         return newFigure;
275     }
276 
277     /**
278      * Obtention de la dernière figure (implicitement celle qui est en cours de
279      * dessin)
280      * @return la dernière figure du dessin
281      */
282     public Figure getLastFigure()
283     {
284         // TODO Remplacer par l'implémentation ...
285         return null;
286     }
287 
288     /**
289      * Obtention de la dernière figure contenant le point p.
290      * @param p le point sous lequel on cherche une figure
291      * @return une référence vers la dernière figure contenant le point p ou à
292      * défaut null.
293      */
294     public Figure getFigureAt(Point2D p)
295     {
296         selectedFigure = null;
297 
298         /*
299          * TODO Recherche dans le flux des figures de la DERNIERE figure
300          * contenant le point p.
301          */
302 
303         return selectedFigure;
304     }
305 
306     /**
307      * Retrait de la dernière figure
308      * @post le modèle de dessin a été mis à jour
309      */
310     public void removeLastFigure()
311     {
312         // TODO Compléter ...
313     }
314 
315     /**
316      * Effacement de toutes les figures (sera déclenché par une action clear)
317      * @post le modèle de dessin a été mis à jour
318      */
319     public void clear()
320     {
321         // TODO Compléter ...
322     }
323 
324     /**
325      * Accesseur de l'état de filtrage
326      * @return l'état courant de filtrage
327      */
328     public boolean getFiltering()
329     {
330         return filtering;
331     }
332 
333     /**
334      * Changement d'état du filtrage
335      * @param filtering le nouveau statut de filtrage
336      * @post le modèle de dessin a été mis à jour
337      */
338     public void setFiltering(boolean filtering)
339     {
340         // TODO ... filtering ...
341     }
342 
343     /**
344      * Ajout d'un filtre pour filtrer les types de figures
345      * @param filter le filtre à ajouter
346      * @return true si le filtre n'était pas déjà présent dans l'ensemble des
347      * filtres filtrant les types de figures, false sinon
348      * @post si le filtre a été ajouté, une mise à jour est déclenchée
349      */
350     // TODO décommenter lorsque prêt
351 //  public boolean addShapeFilter(ShapeFilter filter)
352 //  {
353 //      // TODO ... shapeFilters ...
354 //      return false;
355 //  }
356 
357     /**
358      * Retrait d'un filtre filtrant les types de figures
359      * @param filter le filtre à retirer
360      * @return true si le filtre faisait partie des filtres filtrant les types
```

```
361      * de figure et a été retiré, false sinon.
362      * @post si le filtre a éré retiré, une mise à jour est déclenchée
363      */
364     // TODO décommenter lorsque prêt
365 //  public boolean removeShapeFilter(ShapeFilter filter)
366 //  {
367 //      // TODO ... shapeFilters ...
368 //      return false;
369 //  }
370
371     /**
372      * Mise en place du filtre de couleur de remplissage
373      * @param filter le filtre de couleur de remplissage à appliquer
374      * @post le {@link #fillColorFilter} est mis en place et une mise à jour
375      * est déclenchée
376      */
377     // TODO décommenter lorsque prêt
378 //  public void setFillColorFilter(FillColorFilter<Paint> filter)
379 //  {
380 //      // TODO ... fillColorFilter ...
381 //  }
382
383     /**
384      * Mise en place du filtre de couleur de trait
385      * @param filter le filtre de couleur de trait à appliquer
386      * @post le #edgeColorFilter est mis en place et une mise à jour
387      * est déclenchée
388      */
389     // TODO décommenter lorsque prêt
390 //  public void setEdgeColorFilter(EdgeColorFilter<Paint> filter)
391 //  {
392 //      // TODO ... edgeColorFilter ...
393 //  }
394
395     /**
396      * Ajout d'un filtre pour filtrer les types de ligne des figures
397      * @param filter le filtre à ajouter
398      * @return true si le filtre n'était pas déjà présent dans l'ensemble des
399      * filtres filtrant les types de lignes, false sinon
400      * @post si le filtre a été ajouté, une mise à jour est déclenchée
401      */
402     // TODO décommenter lorsque prêt
403 //  public boolean addLineFilter(LineFilter filter)
404 //  {
405 //      // TODO ... lineFilters ...
406 //      return false;
407 //  }
408
409     /**
410      * Retrait d'un filtre filtrant les types de lignes
411      * @param filter le filtre à retirer
412      * @return true si le filtre faisait partie des filtres filtrant les types
413      * de lignes et a été retiré, false sinon.
414      * @post si le filtre a éré retiré, une mise à jour est déclenchée
415      */
416     // TODO décommenter lorsque prêt
417 //  public boolean removeLineFilter(LineFilter filter)
418 //  {
419 //      // TODO ... lineFilters ...
420 //      return false;
421 //  }
422
423     /**
424      * Remise à l'état non sélectionné de toutes les figures
425      */
426     public void clearSelection()
427     {
428         // TODO Compléter ...
429     }
430
431     /**
432      * Mise à jour des indices des figures sélectionnées dans {@link #selectionIndex}
433      * d'après l'interrogation de l'ensembles des figures (après filtrage).
434      */
435     public void updateSelection()
436     {
437         // TODO Compléter ...
438     }
439
440     /**
441      * Indique s'il existe des figures sélectionnées
442      * @return true s'il y a des figures sélectionnées
443      */
444     public boolean hasSelection()
445     {
446         // TODO Remplacer par l'implémentation
447         return false;
448     }
449
450     /**
```



```
451      * Destruction des figures sélectionnées.
452      * Et incidemment nettoyage de {@link #selectionIndex}
453      */
454     public void deleteSelected()
455     {
456         // TODO Compléter ...
457     }
458
459     /**
460      * Applique un style particulier aux figure sélectionnées
461      * @param fill la couleur de remplissage à applique aux figures sélectionnées
462      * @param edge la couleur de trait à appliquer aux figures sélectionnées
463      * @param stroke le style de trait à appliquer aux figures sélectionnées
464      */
465     public void applyStyleToSelected(Paint fill, Paint edge, BasicStroke stroke)
466     {
467         // TODO Compléter ...
468     }
469
470     /**
471      * Déplacement des figures sélectionnées en haut de la liste des figures.
472      * En conservant l'ordre des figures sélectionnées
473      */
474     public void moveSelectedUp()
475     {
476         // TODO Compléter ...
477
478
479
480         // Mise à jour des index des figures sélectionnées & notif observers
481         updateSelection();
482     }
483
484     /**
485      * Accès aux figures dans un stream afin que l'on puisse y appliquer
486      * de filtres
487      * @return le flux des figures éventuellement filtrés par les différents
488      * filtres
489      */
490     public Stream<Figure> stream()
491     {
492         Stream<Figure> figuresStream = figures.stream();
493         if (filtering)
494         {
495             // TODO Compléter avec ...
496             // if (filters.size() > 0)
497             // {
498             // figuresStream = figuresStream.filter(filters);
499             // }
500         }
501
502         return figuresStream;
503     }
504
505     /* (non-Javadoc)
506      * @see history.Originator#createMemento()
507      */
508     @Override
509     public Memento<Figure> createMemento()
510     {
511         return new Memento<Figure>(figures);
512     }
513
514     /* (non-Javadoc)
515      * @see history.Originator#setMemento(history.Memento)
516      */
517     @Override
518     public void setMemento(Memento<Figure> memento)
519     {
520         if (memento ≠ null)
521         {
522             List<Figure> savedFigures = memento.getState();
523             System.out.println("Drawing::setMemento(" + savedFigures + ")");
524
525             figures.clear();
526             for (Figure elt : savedFigures)
527             {
528                 figures.add(elt.clone());
529             }
530
531             update();
532         }
533         else
534         {
535             System.err.println("Drawing::setMemento(null)");
536         }
537     }
538 }
```

```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.Point;
import java.awt.geom.Point2D;

import javax.swing.JLabel;

import figures.Drawing;
import figures.Figure;
import figures.Rectangle;
import figures.listeners.creation.AbstractCreationListener;
import figures.listeners.creation.RectangularShapeCreationListener;
import history.HistoryManager;

/**
 * Enumeration des différentes figures possibles
 * @author davidroussel
 */
public enum FigureType
{
    /**
     * Les différents types de figures
     */
    CIRCLE, ELLIPSE, RECTANGLE, ROUNDED_RECTANGLE, POLYGON, NGON, STAR;

    /**
     * Nombre de figures référencées ici (à changer si on ajoute des types de
     * figures)
     */
    public final static int NbFigureTypes = 7;

    /**
     * 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 new Circle(stroke, edge, fill, p, 0.0f);
            case ELLIPSE:
                return null; // new Ellipse(stroke, edge, fill, p, p);
            case RECTANGLE:
                return new Rectangle(stroke, edge, fill, p, p);
            case ROUNDED_RECTANGLE:
                return null; // TODO new RoundedRectangle(stroke, edge, fill, p, p, 0);
            case POLYGON:
                Point pp = new Point(Double.valueOf(p.getX()).intValue(),
                                     Double.valueOf(p.getY()).intValue());
                return null; // TODO new Polygon(stroke, edge, fill, pp, pp);
            case NGON:
                return null; // TODO new NGon(stroke, edge, fill, p);
            case STAR:
                return null; // TODO new Star(stroke, edge, fill, p);
        }

        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 history le gestionnaire d'historique pour les Undo/Redo
     * @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,
                                                        HistoryManager<Figure> history,
                                                        JLabel tipLabel)
        throws AssertionError
    {
        switch (this)
        {
            case CIRCLE:
            case ELLIPSE:
```

```
91              case RECTANGLE:
92                  return new RectangularShapeCreationListener(model, history, tipLabel);
93              case ROUNDED_RECTANGLE:
94                  return null; // TODO new RoundedRectangleCreationListener(model, history, tipLabel);
95              case POLYGON:
96                  return null; // TODO new PolygonCreationListener(model, history, tipLabel);
97              case NGON:
98                  return null; // TODO new NGonCreationListener(model, history, tipLabel);
99              case STAR:
100                 return null; // TODO StarCreationListener(model, history, tipLabel);
101         }
102
103         throw new AssertionError("FigureType unknown assertion: " + this);
104     }
105
106     /**
107      * Représentation sous forme de chaine de caractères
108      * @return une chaine de caractère représentant la valeur de cet enum
109      * @throws AssertionError si la valeur de cet enum n'est pas prévue
110      */
111     @Override
112     public String toString() throws AssertionError
113     {
114         switch (this)
115         {
116             case CIRCLE:
117                 return new String("Circle");
118             case ELLIPSE:
119                 return new String("Ellipse");
120             case RECTANGLE:
121                 return new String("Rectangle");
122             case ROUNDED_RECTANGLE:
123                 return new String("Rounded Rectangle");
124             case POLYGON:
125                 return new String("Polygon");
126             case NGON:
127                 return new String("Ngon");
128             case STAR:
129                 return new String("Star");
130         }
131
132         throw new AssertionError("FigureType unknown assertion: " + this);
133     }
134
135     /**
136      * Otention d'un tableau de chaine de caractères contenant l'ensemble des
137      * nom des figures
138      * @return un tableau de chaine de caractères contenant l'ensemble des nom
139      * des figures
140      */
141     public static String[] stringValues()
142     {
143         FigureType[] values = FigureType.values();
144         String[] stringValues = new String[values.length];
145
146         for (int i = 0; i < stringValues.length; i++)
147         {
148             stringValues[i] = values[i].toString();
149         }
150
151         return stringValues;
152     }
153
154     /**
155      * Conversion d'un entier en FigureType
156      * @param i l'entier à convertir en FigureType
157      * @return le FigureType correspondant à l'entier
158      */
159     public static FigureType fromInteger(int i)
160     {
161         switch (i)
162         {
163             case 0:
164                 return CIRCLE;
165             case 1:
166                 return ELLIPSE;
167             case 2:
168                 return RECTANGLE;
169             case 3:
170                 return ROUNDED_RECTANGLE;
171             case 4:
172                 return POLYGON;
173             case 5:
174                 return NGON;
175             case 6:
176                 return STAR;
177             default:
178                 return POLYGON;
179         }
180     }
```



```
181     /**
182      * Conversion en entier d'un type de figure
183      * @return un entier correspondant à l'index du type de figure
184      */
185     public int intValue() throws AssertionError
186     {
187         switch (this)
188         {
189             case CIRCLE:
190                 return 0;
191             case ELLIPSE:
192                 return 1;
193             case RECTANGLE:
194                 return 2;
195             case ROUNDED_RECTANGLE:
196                 return 3;
197             case POLYGON:
198                 return 4;
199             case NGON:
200                 return 5;
201             case STAR:
202                 return 6;
203         }
204
205         throw new AssertionError("FigureType unknown assertion: " + this);
206     }
207 }
```

```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;
    }
}
```

```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);
    }

}
```



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

```java
package figures.listeners;

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

import javax.swing.JLabel;

import figures.Drawing;
import figures.Figure;
import history.HistoryManager;

/**
 * Listener (incomplet) des évènements souris pour agir sur lesfigures.
 * Chaque action sur les figures (création ou transformation) est graphiquement
 * construite par une suite de pressed/drag/release ou de clicks qui peut être
 * différente pour chaque type d'action. Aussi les classes filles devront
 * implémenter leur propre xxxFigureListener assurant la gestion des évènement
 * souris
 * @author davidroussel
 */
public abstract class AbstractFigureListener
    implements MouseListener, MouseMotionListener, MouseWheelListener
{
    /**
     * 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;

    /**
     * L'History manager qui gère les historiques d'Undo et de Redo
     */
    protected HistoryManager<Figure> history;

    /**
     * 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
     */
    protected int currentStep;
```



```java

    /**
     * Constructeur protégé (destiné à être utilisé par les classes filles)
     * @param model le modèle de dessin à modifier par ce listener
     * @param history le gestionnaire d'historique pour créer des sauvegardes
     * de l'état courant des figures avant toute modification des figures
     * @param infoLabel le label dans lequel afficher les conseils d'utilisation
     * @param nbSteps le nombres d'étapes de l'action à réaliser
     */
    protected AbstractFigureListener(Drawing model,
                                     HistoryManager<Figure> history,
                                     JLabel infoLabel,
                                     int nbSteps)
    {
        drawingModel = model;
        this.history = history;
        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("AbstractFigureListener caution null "
                + "drawing model");
        }

        if (history == null)
        {
            System.err.println("AbstractFigureListener caution null "
                + "history manager");
        }

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

    /**
     * Initialisation de l'action :
     * Détermine le point de départ ({@link #startPoint})
     * Les classes filles devront réutiliser cette méthode pour récupérer le
     * point de départ de l'action. Puis elles devront initier l'action
     * et enfin passer à l'étape suivante (éventuellement en mettant à jour
     * le modèle dessin.
     * Passe à l'étape suivante avec {@link #nextStep()} ce qui met à jour
     * le {@link #tipLabel}.
     * Met à jour le modèle de dessin avec {@link Drawing#update()}
     * A utiliser dans {@link MouseListener#mousePressed(MouseEvent)} ou bien
     * dans {@link MouseListener#mouseClicked(MouseEvent)} suivant l'action à
     * réaliser.
     * @param e l'évènement souris à utiliser pour initier la création d'une
     * nouvelle figure à la position de cet évènement
     */
    public abstract void startAction(MouseEvent e);

    /**
     * Terminaison de l'action sur une figure:
     * remet l'étape courante à 0 en passant à l'étape suivante (ce qui met à
     * jour le {@link #tipLabel} avec {@link #updateTip()}).
     * détermine la position du point de terminaison de la figure
     * ({@link #endPoint}), puis met à jour le dessin ({@link Drawing#update()}).
     * 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 abstract void endAction(MouseEvent e);

    /**
     * Récupération du point de départ de l'action
     * @param e l'évènement souris d'où l'on veut récupérer le point de départ
     */
    public void setStartPoint(MouseEvent e)
    {
        startPoint = e.getPoint();
    }

    /**
     * Récupération du point de terminaison de l'action
     * @param e l'évènement souris d'où l'on veut récupérer le point de terminaison
     */
    public void setendPoint(MouseEvent e)
    {
        endPoint = e.getPoint();
```

```
181     }
182
183     /**
184      * Passage à l'étape suivante et mise à jours des conseils utilisateurs
185      * relatifs à l'étape suivante.
186      * Lorsque le passage à l'étape suivante dépasse le nombre d'étapes prévues
187      * l'étape courante est remise à 0.
188      * @see #currentStep
189      * @see #updateTip()
190      */
191     protected void nextStep()
192     {
193         if (currentStep < (nbSteps - 1))
194         {
195             currentStep++;
196         }
197         else
198         {
199             currentStep = 0;
200         }
201
202 //      System.out.println(getClass().getSimpleName() + " nextStep to step "
203 //          + currentStep);
204
205         updateTip();
206     }
207
208     /**
209      * Mise à jour du conseil dans le {@link #tipLabel} en fonction de l'étape
210      * courante
211      */
212     protected void updateTip()
213     {
214         if (tipLabel ≠ null)
215         {
216             tipLabel.setText(tips[currentStep]);
217         }
218         else
219         {
220             System.err.println(getClass().getSimpleName() + "::updateTip : null tipLabel");
221         }
222     }
223 }
```



```
1   /**
2    *
3    */
4   package figures.listeners;
5
6   import java.awt.event.MouseEvent;
7   import java.awt.event.MouseWheelEvent;
8
9   import javax.swing.JLabel;
10
11  import figures.Drawing;
12  import figures.Figure;
13  import history.HistoryManager;
14
15  /**
16   * Listener permettant d'ajouter ou de retirer des figures de la liste des
17   * figures sélectionnées
18   * @author davidroussel
19   */
20  public class SelectionFigureListener extends AbstractFigureListener
21  {
22
23      /**
24       * Constructeur
25       * @param model le modèle de dessin sur lequel on opère
26       * @param history le gestionnaire d'historique pour les Undo/Redo
27       * @param infoLabel le label dans lequel afficher les conseils d'utilisation
28       */
29      public SelectionFigureListener(Drawing model,
30                                     HistoryManager<Figure> history,
31                                     JLabel infoLabel)
32      {
33          super(model, history, infoLabel, 1);
34
35          tips[0] = new String("Cliquez pour sélectionner/déselectionner une figure");
36          updateTip();
37      }
38
39      /* (non-Javadoc)
40       * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
41       */
42      @Override
43      public void mouseClicked(MouseEvent e)
44      {
45          nextStep(); // inutile
46
47          // S'il y a une figure sous le curseur on l'ajoute où on l'enlève
48          // de la sélection suivant son état courant de sélection
49          currentFigure = drawingModel.getFigureAt(e.getPoint());
50
51          if (currentFigure ≠ null)
52          {
53              currentFigure.setSelected(¬currentFigure.isSelected());
54
55              drawingModel.updateSelection();
56          }
57      }
58
59      /* (non-Javadoc)
60       * @see java.awt.event.MouseListener#mousePressed(java.awt.event.MouseEvent)
61       */
62      @Override
63      public void mousePressed(MouseEvent e)
64      {
65          // Rien
66      }
67
68      /* (non-Javadoc)
69       * @see java.awt.event.MouseListener#mouseReleased(java.awt.event.MouseEvent)
70       */
71      @Override
72      public void mouseReleased(MouseEvent e)
73      {
74          // Rien
75      }
76
77      /* (non-Javadoc)
78       * @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
79       */
80      @Override
81      public void mouseEntered(MouseEvent e)
82      {
83          // Rien
84      }
85
86      /* (non-Javadoc)
87       * @see java.awt.event.MouseListener#mouseExited(java.awt.event.MouseEvent)
88       */
89      @Override
90      public void mouseExited(MouseEvent e)
```

```
91      {
92          // Rien
93      }
94
95      /* (non-Javadoc)
96       * @see java.awt.event.MouseMotionListener#mouseDragged(java.awt.event.MouseEvent)
97       */
98      @Override
99      public void mouseDragged(MouseEvent e)
100     {
101         // Rien
102     }
103
104     /* (non-Javadoc)
105      * @see java.awt.event.MouseMotionListener#mouseMoved(java.awt.event.MouseEvent)
106      */
107     @Override
108     public void mouseMoved(MouseEvent e)
109     {
110         // Rien
111     }
112
113     /* (non-Javadoc)
114      * @see java.awt.event.MouseWheelListener#mouseWheelMoved(java.awt.event.MouseWheelEvent)
115      */
116     @Override
117     public void mouseWheelMoved(MouseWheelEvent e)
118     {
119         // Rien
120     }
121
122     /* (non-Javadoc)
123      * @see figures.listeners.AbstractFigureListener#startAction(java.awt.event.MouseEvent)
124      */
125     @Override
126     public void startAction(MouseEvent e)
127     {
128         // Rien
129     }
130
131     /* (non-Javadoc)
132      * @see figures.listeners.AbstractFigureListener#endAction(java.awt.event.MouseEvent)
133      */
134     @Override
135     public void endAction(MouseEvent e)
136     {
137         // Rien
138     }
139 }
```



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

```java
package figures.listeners.creation;

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.Drawing;
import figures.Figure;
import figures.listeners.AbstractFigureListener;
import history.HistoryManager;

/**
 * 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 extends AbstractFigureListener
    implements MouseListener, MouseMotionListener
{
    /**
     * Constructeur protégé (destiné à être utilisé par les classes filles)
     * @param model le modèle de dessin à modifier par ce creationListener
     * @param history le gestionnaire d'historique pour les Undo/Redo
     * @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,
                                       HistoryManager<Figure> history,
                                       JLabel infoLabel,
                                       int nbSteps)
    {
        super(model, history, infoLabel, nbSteps);
    }

    /**
     * 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
     */
    @Override
    public void startAction(MouseEvent e)
    {
        history.record();
        setStartPoint(e);
        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
     */
    @Override
    public void endAction(MouseEvent e)
    {
        // Remise à zéro de currentStep pour pouvoir réutiliser ce
        // listener sur une autre figure
        nextStep();

        setendPoint(e);

        // à la fin de la figure on la normalise pour qu'elle soit centrée
        // sur son barycentre et la position du barycentre dans la translation
        if (currentFigure != null)
        {
            currentFigure.normalize();
        }
        else
        {
            System.err.println(getClass().getSimpleName() + "::endAction : null figure");
        }

        if (checkZeroSizeFigure())
        {
            // cancel last memento
            history.cancel();
        }

        drawingModel.update();

        updateTip();
    }

    /**
     * Contrôle de la taille de la figure créée à effectuer à la fin de la
     * création afin d'éliminer les figures de taille 0.
     * @return true si une figure de petite taille a été retirée
     * @see #startPoint
     * @see #endPoint
     */
    protected boolean checkZeroSizeFigure()
    {
        if (startPoint.distance(endPoint) < 1.0)
        {
            drawingModel.removeLastFigure();
            System.err.println("Removed zero sized figure");
            return true;
        }

        return false;
    }
}
```

```java
package figures.listeners.creation;

import java.awt.event.MouseEvent;
import java.awt.event.MouseWheelEvent;

import javax.swing.JLabel;

import figures.Drawing;
import figures.Figure;
import history.HistoryManager;

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

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

        updateTip();

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

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

    /**
     * Terminaison de la nouvelle figure rectangulaire si le bouton appuyé
     * était le bouton gauche
     * @param e l'évènement souris
     * @see AbstractCreationListener#endAction(MouseEvent)
     * @see java.awt.event.MouseListener#mouseReleased(java.awt.event.MouseEvent)
     */
    @Override
    public void mouseReleased(MouseEvent e)
    {
        if ((e.getButton() == MouseEvent.BUTTON1) && (currentStep == 1))
        {
            endAction(e);
        }
    }

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

    /* (non-Javadoc)
     * @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
     */
```



```java
    @Override
    public void mouseEntered(MouseEvent e)
    {
        // Rien
    }

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

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

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

            drawingModel.update();
        }
    }


    /* (non-Javadoc)
     * @see java.awt.event.MouseWheelListener#mouseWheelMoved(java.awt.event.MouseWheelEvent)
     */
    @Override
    public void mouseWheelMoved(MouseWheelEvent e)
    {
        // Rien

    }
}
```

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



```java
package figures.listeners.transform;

import java.awt.event.InputEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseWheelEvent;
import java.awt.geom.AffineTransform;
import java.awt.geom.Point2D;

import javax.swing.JLabel;

import figures.Drawing;
import figures.Figure;
import figures.listeners.AbstractFigureListener;
import figures.listeners.creation.AbstractCreationListener;
import history.HistoryManager;

/**
 * Listener permettant de transformer une figure
 * <ol>
 * <li>bouton 1 pressé et maintenu enfoncé</li>
 * <li>déplacement de la souris avec le bouton enfoncé</li>
 * <li>relachement du bouton</li>
 * </ol>
 * @author davidroussel
 */
public abstract class AbstractTransformShapeListener extends AbstractFigureListener
{
    /**
     * La transformation initiale de la figure
     */
    protected AffineTransform initialTransform;

    /**
     * Indique si seules les figures sélectionnées sont transformables ou pas
     */
    protected boolean onlySelected;

    /**
     * Le centre de la figure sélectionnée (car on l'utilisera souvent)
     */
    protected Point2D center;

    /**
     * Le modificateur (Ctrl, Shift, Alt, etc.) applicable lors du traitement
     * des évènements souris
     * @see InputEvent#SHIFT_DOWN_MASK
     * @see InputEvent#CTRL_DOWN_MASK
     * @see InputEvent#ALT_DOWN_MASK
     * @see InputEvent#META_DOWN_MASK
     */
    protected int keyMask;

    /**
     * Valeur par défaut lorsqu'aucun key mask n'est requis
     */
    protected static final int NoKeyMask = 0;

    /**
     * Constructeur d'un listener à deux étapes: pressed->drag->release pour
     * transformer les figures
     * @param model le modèle de dessin à modifier par ce Listener
     * @param history le gestionnaire d'historique
     * @param tipLabel le label dans lequel afficher les conseils utilisateur
     */
    public AbstractTransformShapeListener(Drawing model,
                                          HistoryManager<Figure> history,
                                          JLabel tipLabel)
    {
        super(model, history, tipLabel, 2);

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

        updateTip();

        System.out.println(getClass().getSimpleName() + " created");

        center = null;

        keyMask = NoKeyMask;
    }

    /**
     * Vérifie que seul le {@link InputEvent#BUTTON1_MASK} ainsi que le
     * {@link #keyMask} sont présents dans les modifiers renvoyés par
     * {@link MouseEvent#getModifiers()} mais <b>aucun autre</b> modifier
     * @param modifiers les modifiers à vérifier
     * @return true si seuls {@link InputEvent#BUTTON1_MASK} et {@link #keyMask}
     * sont présents dans les modifiers, false sinon
```

```
91      */
92      public boolean checkModifiers(int modifiers)
93      {
94          return modifiers ≡ (InputEvent.BUTTON1_MASK | keyMask);
95      }
96
97      /**
98       * Initialisation de la transformation de la figure. Détermine le point de
99       * départ de la transformation de la figure ({@link #startPoint}) ainsi que
100      * la figure sélectionnée qui peut éventuellement être nulle s'il n'y a pas
101      * de figures sélectionnées ou sous le curseur.
102      * A utiliser dans
103      * {@link MouseListener#mousePressed(MouseEvent)} ou bien dans
104      * {@link MouseListener#mouseClicked(MouseEvent)} suivant la figure à créer.
105      * @see #mousePressed(MouseEvent)
106      * @see #mouseClicked(MouseEvent)
107      */
108     @Override
109     public void startAction(MouseEvent e)
110     {
111         history.record();
112
113         setStartPoint(e);
114
115         currentFigure = drawingModel.getFigureAt(startPoint);
116         if (currentFigure ≠ null)
117         {
118             center = currentFigure.getCenter();
119
120             init();
121
122             nextStep();
123
124             drawingModel.update(); // optionel
125         }
126         else
127         {
128             System.err.println(getClass().getSimpleName() + "::startAction : null figure");
129         }
130     }
131
132     /**
133      * Initialisations particulières à l'initialisation du listener
134      * <ul>
135      *   <li>Initialisation de transformation initiale</li>
136      *   <li>...</li>
137      * </ul>
138      */
139     public abstract void init();
140
141     /**
142      * Terminaison du déplacement d'une figure, remet l'étape courante à 0,
143      * détermine la position du point de terminaison du déplacement de la figure
144      * ({@link #endPoint}), puis met à jour le dessin (
145      * {@link Drawing#update()}) et les conseils utilisateurs (
146      * {@link #updateTip()}). A utiliser dans un
147      * {@link MouseListener#mousePressed(MouseEvent)} ou bien dans un
148      * {@link MouseListener#mouseClicked(MouseEvent)} suivant la figure à créer.
149      * @param e l'évènement souris à utiliser lors de la terminaison d'un figure
150      */
151     @Override
152     public void endAction(MouseEvent e)
153     {
154         if (currentStep ≡ 1)
155         {
156             // Remise à zéro de currentStep pour pouvoir réutiliser ce
157             // listener sur une autre figure
158             nextStep();
159
160             setendPoint(e);
161
162             currentFigure = null;
163
164             drawingModel.update();
165         }
166     }
167
168     /**
169      * Création d'une nouvelle figure rectangulaire de taille 0 au point de
170      * l'évènement souris, si le bouton appuyé est le bouton gauche.
171      * @param e l'évènement souris
172      * @see AbstractCreationListener#startAction(MouseEvent)
173      * @see java.awt.event.MouseListener#mousePressed(java.awt.event.MouseEvent)
174      */
175     @Override
176     public void mousePressed(MouseEvent e)
177     {
178         currentFigure = drawingModel.getFigureAt(e.getPoint());
179
180         if (currentFigure ≠ null)
```



```
181         {
182             if (currentFigure.isSelected() ∧
183 //              (e.getButton() == MouseEvent.BUTTON1) &&
184                 checkModifiers(e.getModifiers()))
185             {
186                 startAction(e);
187             }
188         }
189         else
190         {
191             System.err.println(getClass().getSimpleName() + "::mousePressed : null figure");
192         }
193     }
194
195     /**
196      * Terminaison de la nouvelle figure rectangulaire si le bouton appuyé
197      * était le bouton gauche
198      * @param e l'évènement souris
199      * @see AbstractCreationListener#endAction(MouseEvent)
200      * @see java.awt.event.MouseListener#mouseReleased(java.awt.event.MouseEvent)
201      */
202     @Override
203     public void mouseReleased(MouseEvent e)
204     {
205         if (e.getButton() ≡ MouseEvent.BUTTON1) // On se fiche du keymask pour terminer l'action
206         {
207             // System.out.println("TransformShapeListener ended...");
208             endAction(e);
209         }
210     }
211
212     /*
213      * (non-Javadoc)
214      * @see java.awt.event.MouseListener#mouseClicked(java.awt.event.MouseEvent)
215      */
216     @Override
217     public void mouseClicked(MouseEvent e)
218     {
219         // Rien
220     }
221
222     /*
223      * (non-Javadoc)
224      * @see java.awt.event.MouseListener#mouseEntered(java.awt.event.MouseEvent)
225      */
226     @Override
227     public void mouseEntered(MouseEvent e)
228     {
229         // Rien
230     }
231
232     /*
233      * (non-Javadoc)
234      * @see java.awt.event.MouseListener#mouseExited(java.awt.event.MouseEvent)
235      */
236     @Override
237     public void mouseExited(MouseEvent e)
238     {
239         // Rien
240     }
241
242     /*
243      * (non-Javadoc)
244      * @see
245      * java.awt.event.MouseMotionListener#mouseMoved(java.awt.event.MouseEvent)
246      */
247     @Override
248     public void mouseMoved(MouseEvent e)
249     {
250         // Rien
251     }
252
253     /**
254      * Déplacement du point en bas à droite de la figure rectangulaire, si
255      * l'on se trouve à l'étape 1 (après initalisation du déplacement) et que
256      * le bouton enfoncé est bien le bouton gauche.
257      * @see java.awt.event.MouseMotionListener#mouseDragged(java.awt.event.MouseEvent)
258      */
259     @Override
260     public void mouseDragged(MouseEvent e)
261     {
262         if (currentStep ≡ 1)
263         {
264             if (currentFigure ≠ null)
265             {
266                 updateDrag(e);
267
268                 drawingModel.update();
269             }
270             else
```

```
271                 {
272                     System.err.println(getClass().getSimpleName() + "::mouseDragged : null figure");
273                 }
274             }
275     }
276
277     /**
278      * Mise à jour de la transformation courante et application
279      * de la transformation initiale ({@link #initialTransformation} et
280      * de la transformation courante
281      * @param e évènement souris
282      */
283     public abstract void updateDrag(MouseEvent e);
284
285     /* (non-Javadoc)
286      * @see java.awt.event.MouseWheelListener#mouseWheelMoved(java.awt.event.MouseWheelEvent)
287      */
288     @Override
289     public void mouseWheelMoved(MouseWheelEvent e)
290     {
291         // Rien
292     }
293 }
```



```
1  package figures.listeners.transform;
2
3  import java.awt.event.MouseEvent;
4  import java.awt.geom.AffineTransform;
5  import java.awt.geom.Point2D;
6
7  import javax.swing.JLabel;
8
9  import figures.Drawing;
10 import figures.Figure;
11 import history.HistoryManager;
12
13 /**
14  * Listener permettant de déplacer une figure
15  * <ol>
16  *  <li>bouton 1 pressé et maintenu enfoncé</li>
17  *  <li>déplacement de la souris avec le bouton enfoncé</li>
18  *  <li>relachement du bouton</li>
19  * </ol>
20  * @author davidroussel
21  */
22 public class MoveShapeListener extends AbstractTransformShapeListener
23 {
24     /**
25      * Le dernier point
26      * @note Utilisé pour calculer le déplacement entre l'évènement courant
27      * et l'évènement précédent
28      * @note modifié dans {@link #mouseDragged(MouseEvent)}
29      */
30     private Point2D lastPoint;
31
32     /**
33      * Constructeur d'un listener à deux étapes: pressed->drag->release pour
34      * déplacer toutes les figures
35      * @param model le modèle de dessin à modifier par ce Listener
36      * @param history le gestionnaire d'historique pour les Undo/Redo
37      * @param tipLabel le label dans lequel afficher les conseils utilisateur
38      */
39     public MoveShapeListener(Drawing model,
40                              HistoryManager<Figure> history,
41                              JLabel tipLabel)
42     {
43         super(model, history, tipLabel);
44     }
45
46     /* (non-Javadoc)
47      * @see figures.listeners.transform.AbstractTransformShapeListener#init()
48      */
49     @Override
50     public void init()
51     {
52         lastPoint = startPoint;
53         if (currentFigure ≠ null)
54         {
55             initialTransform = currentFigure.getTranslation();
56             // System.out.println("MoveShapeListener2 initialized");
57         }
58         else
59         {
60             System.err.println(getClass().getSimpleName() + "::init : null figure");
61         }
62     }
63
64     /* (non-Javadoc)
65      * @see figures.listeners.transform.AbstractTransformShapeListener#updateDrag(java.awt.event.Mou
   seEvent)
66      */
67     @Override
68     public void updateDrag(MouseEvent e)
69     {
70         // System.out.println("MoveShapeListener2 dragged");
71         Point2D currentPoint = e.getPoint();
72         double dx = currentPoint.getX() - lastPoint.getX();
73         double dy = currentPoint.getY() - lastPoint.getY();
74         AffineTransform translate = AffineTransform.getTranslateInstance(dx, dy);
75         translate.concatenate(initialTransform);
76         currentFigure.setTranslation(translate);
77     }
78 }
```

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



```java
package figures.treemodels;

import java.util.HashSet;
import java.util.List;
import java.util.Observable;
import java.util.Observer;
import java.util.Set;
import java.util.Vector;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import javax.swing.JTree;
import javax.swing.event.TreeModelEvent;
import javax.swing.event.TreeModelListener;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.TreeModel;
import javax.swing.tree.TreePath;
import javax.swing.tree.TreeSelectionModel;

import figures.Drawing;
import figures.Figure;
/**
 * Classe abstraite de base de tous les arbres composés de figures
 * @author davidroussel
 */
public abstract class AbstractFigureTreeModel implements TreeModel, Observer, TreeSelectionListener
{
    /**
     * L'élément racine de l'arbre (une simple chaine de caractères)
     */
    protected String rootElement;

    /**
     * Le modèle de dessin.
     * On a besoin de garder une référence vers le modèlde de dessin lorsque
     * la liste des figures sélectionnées dans l'arbre change afin que l'on
     * puisse le notifier des changements
     */
    protected Drawing drawing;

    /**
     * Le JTree utilisé pour visualiser cet arbre.
     * On a besoin de garder une référence vers cette vue afin de
     * spécifier (programmatiquement) quels sont les noeuds sélectionnés
     * en fonction des figures sélectionnées.
     * @see #selectedFigures
     */
    protected JTree treeView;

    /**
     * Liste des figures sélectionnées dans l'arbre
     */
    protected Set<TreePath> selectedFigures;

    /**
     * La liste des listeners de ce modèle
     */
    protected Vector<TreeModelListener> treeModelListeners;

    /**
     * Indique si un évènement est généré à l'intérieur du TreeModel ou
     * bien s'il provient de l'UI
     */
    protected boolean selfEvent;

    /**
     * Constructeur de l'arbre des figures
     * @param drawing le modèle de dessin
     * @param tree le {@link JTree} utilisé pour visualiser cet arbre
     * @param rootName le nom du noeud racine
     */
    public AbstractFigureTreeModel(Drawing drawing, JTree tree, String rootName)
        throws NullPointerException
    {
        this.drawing = drawing;
        treeView = tree;

        rootElement = new String(rootName);
        selectedFigures = new HashSet<TreePath>();
        treeModelListeners = new Vector<TreeModelListener>();

        if (this.drawing ≠ null)
        {
            this.drawing.addObserver(this);
        }
        else
        {
            throw new NullPointerException("AbstractFigureTreeModel(null drawing)");
        }
```

```
91
92          if (treeView ≠ null)
93          {
94              treeView.setModel(this);
95              treeView.addTreeSelectionListener(this);
96          }
97          else
98          {
99              throw new NullPointerException("AbstractFigureTreeModel(null tree)");
100         }
101
102         selfEvent = false;
103     }
104
105     /**
106      * Nettoyage avant destruction
107      */
108     @Override
109     protected void finalize() throws Throwable
110     {
111         drawing.deleteObserver(this);
112         rootElement = null;
113         drawing = null;
114         treeView.removeTreeSelectionListener(this);
115         treeView = null;
116         selectedFigures.clear();
117         selectedFigures = null;
118         treeModelListeners.clear();
119         treeModelListeners = null;
120         super.finalize();
121     }
122
123     /**
124      * Mise à jour par l'observable (en l'occurrence un {@link Drawing})
125      * @param observable le {@link Drawing}
126      * @param data les données à transmettre (non utilisé ici)
127      * @see Observer#update(Observable, Object)
128      */
129     @Override
130     public void update(Observable observable, Object data)
131     {
132         if (observable instanceof Drawing)
133         {
134             synchronized (observable)
135             {
136                 drawing = (Drawing) observable;
137                 Stream<Figure> stream = drawing.stream();
138
139                 // Obtention d'une collection de figures à dessiner
140                 Vector<Figure> figures = stream.sequential()
141                     .collect(Collectors.toCollection(Vector::new));
142
143                 // Effacement des chemins des figures sélectionnées
144                 selectedFigures.clear();
145
146                 // Mise à jour de l'arbre des figures
147                 updateFiguresFromDrawing(figures);
148
149                 // Mise à jour des chemins des figures sélectionnées
150                 updateSelectedFigures();
151
152                 // Mise à jour des figures sélectionnées dans le treeView
153                 updateSelectedPath();
154             }
155         }
156         else
157         {
158             System.err.println("Observable is not an instance of Drawing");
159         }
160     }
161
162     /**
163      * Mise à jour des figures de l'arbre en les comparant une par une aux
164      * figures du modèle de dessin.
165      * Permet l'enlever/ajouter les figures de l'arbre en fonction des
166      * modifications observées dans les figures du modèle de dessin.
167      * @param figures les figures du modèle de dessin
168      */
169     protected abstract void updateFiguresFromDrawing(List<Figure> figures);
170
171     /**
172      * Mise à jour de {@link #selectedFigures} d'après les figures de l'arbre
173      * sélectionnées.
174      */
175     protected abstract void updateSelectedFigures();
176
177     /**
178      * Mise à jour des noeuds sélectionnés dans le {@link #treeView} d'après
179      * les paths répertoriés dans {@link #selectedFigures}
180      */
```



```
181     protected void updateSelectedPath()
182     {
183         if (treeView ≠ null)
184         {
185             TreeSelectionModel tsm = treeView.getSelectionModel();
186             if (tsm ≠ null)
187             {
188                 TreePath[] treePathes = selectedFigures.toArray(new TreePath[0]);
189                 if(treePathes.length ≡ 0)
190                 {
191                     treePathes = null; // pour effacer la sélection
192                 }
193                 tsm.setSelectionPaths(treePathes);
194             }
195             else
196             {
197                 System.err.println("AbstractFigureTreeModel::updateSelectedPath : null Selection Model");
198             }
199         }
200         else
201         {
202             System.err.println("AbstractFigureTreeModel::updateSelectedPath : null TreeView");
203         }
204     }
205
206     /**
207      * Méthode à utiliser lorsque la structure de l'arbre change.
208      * Tous les éléments situés en dessous de path sont mis à jour
209      * @param path le chemin en dessous duquel l'arbre a changé
210      */
211     protected synchronized void fireTreeStructureChanged(TreePath path)
212     {
213         if (treeModelListeners.size() > 0)
214         {
215             /*
216              * Used to create an event when the node structure has changed in
217              * some way, identifying the path to the root of the modified
218              * subtree as a TreePath object.
219              */
220             TreeModelEvent e = new TreeModelEvent(this, path);
221             for (TreeModelListener tml : treeModelListeners)
222             {
223                 selfEvent = true;
224 //              System.out.println("fireTreeStructureChanged(" + e + " to " + tml);
225                 tml.treeStructureChanged(e);
226             }
227         }
228     }
229
230     /**
231      * Méthode à utliser lorsqu'un ou plusieurs noeuds sont ajoutés à
232      * l'arbre
233      * @param path the path to the parent of inserted node(s)
234      * @param newchildIndices an array of the indices of the new inserted nodes
235      * @param newNodes an array of the new inserted nodes (Optional)
236      * @see javax.swing.event.TreeModelListener#treeNodesInserted(TreeModelEvent)
237      */
238     protected synchronized void fireTreeNodesInserted(TreePath path,
239                                                       int[] newchildIndices,
240                                                       Object[] newNodes)
241     {
242         if (treeModelListeners.size() > 0)
243         {
244             TreeModelEvent e =
245                 new TreeModelEvent(this, path, newchildIndices, newNodes);
246             for (TreeModelListener tml : treeModelListeners)
247             {
248                 selfEvent = true;
249 //              System.out.println("fireTreeNodesInserted(" + e + " to " + tml);
250                 tml.treeNodesInserted(e);
251             }
252         }
253     }
254
255     /**
256      * Méthode à utiliser lorsqu'un ou plusieurs noeuds sont retirés de l'arbre
257      * @param path the path to the former parent of deleted node
258      * @param oldChildIndices an array of indices (in ascending order) where
259      * the removed nodes used to be
260      * @note if a subtree is removed from the tree, this method may only be
261      * invoked once for the root of the removed subtree, not once for
262      * each individual set of siblings removed.
263      */
264     protected synchronized void fireTreeNodesRemoved(TreePath path,
265                                                      int[] oldChildIndices,
266                                                      Object[] oldNodes)
267     {
268         if (treeModelListeners.size() > 0)
269         {
270             TreeModelEvent e = new TreeModelEvent(this,
```

```
                                                path,
                                                oldChildIndices,
                                                oldNodes);
                for (TreeModelListener tml : treeModelListeners)
                {
                    selfEvent = true;
//                  System.out.println("fireTreeNodesRemoved(" + e + " to " + tml);
                    tml.treeNodesRemoved(e);
                }
            }
        }

        /**
         * Méthode à utiliser lorsqu'un ou plusieurs noeuds sont changés (par
         * exemple s'il sont sélectionnés programmatiquement)
         * @param treePathes l'ensemble des {@link TreePath} des noeuds changés
         */
        protected synchronized void fireNodesChanged(TreePath[] treePathes)
        {
            for (int i = 0; i < treePathes.length; i++)
            {
                for (TreeModelListener tml : treeModelListeners)
                {
                    selfEvent = true;
//                  System.out.println("fireNodesChanged(" + treePathes[i] + " to " + tml);
                    tml.treeNodesChanged(new TreeModelEvent(this, treePathes[i]));
                }
            }
        }

        /**
         * Accès au noeud d'index index fils du noeud de parent
         * @see javax.swing.tree.TreeModel#getChild(java.lang.Object, int)
         * @param parent le noeud parent du noeud recherché
         * @param index l'index du noeud enfant recherché
         * @return le noeud recherché ou bien null s'il n'existe pas.
         */
        @Override
        public abstract Object getChild(Object parent, int index);

        /**
         * Nombre d'enfants d'un noeud
         * @param parent le noeud dont on veut connaitre le nombre d'enfants.
         * @return le nombre d'enfants du noeud ou bien 0 si ce noeud n'a pas
         * d'enfants ou est une feuille de l'arbre
         * @see javax.swing.tree.TreeModel#getChildCount(java.lang.Object)
         */
        @Override
        public abstract int getChildCount(Object parent);

        /**
         * Index d'un enfant particulier à partir d'un noeud parent
         * @param parent le noeud parent
         * @param child le noeud enfant
         * @return l'index du noeud enfant dans le noeud parent, si parent ou
         * child sont null, ou si l'un des deux n'est pas un noeud de cet arbre
         * renvoie -1.
         * @see javax.swing.tree.TreeModel#getIndexOfChild(java.lang.Object,
         * java.lang.Object)
         */
        @Override
        public abstract int getIndexOfChild(Object parent, Object child);

        /**
         * Accesseur à la racine de l'arbre
         * @return la racine de l'arbre
         */
        @Override
        public Object getRoot()
        {
            return rootElement;
        }

        /**
         * Indique si un noeud est une feuille de l'arbre
         * @param node le noeud dont on veut savoir s'il est une feuille
         * @return true si le noeud est une feuille de l'arbre, false autrement.
         * @see javax.swing.tree.TreeModel#isLeaf(java.lang.Object)
         */
        @Override
        public abstract boolean isLeaf(Object node);

        /**
         * Méthode déclenchée lorsqu'un utilisateur a altéré la valeur d'un item
         * identifié par path avec la nouvelle valeur newValue. Si newValue est
         * effectivement une nouvelle valeur, alors on doit déclencher un
         * treeNodesChanged event. [Non utilisé ici]
         * @param path le chemin du noeud modifié
         * @param newValue la nouvelle valeur du noeud
         */
```



```
        @Override
        public void valueForPathChanged(TreePath path, Object newValue)
        {
            System.out.println("*** valueForPathChanged : " + path + " --> " +
                               newValue);
        }

        /**
         * Ajout d'un listener à ce modèle d'abre
         * @param l le listener à ajouter
         */
        @Override
        public void addTreeModelListener(TreeModelListener l)
        {
            if ((l ≠ null) ∧ ¬treeModelListeners.contains(l))
            {
                treeModelListeners.add(l);
            }
        }

        /**
         * Retrait d'n listener à ce modèle d'arbre
         * @param l le listener à retirer
         */
        @Override
        public void removeTreeModelListener(TreeModelListener l)
        {
            if (treeModelListeners.contains(l))
            {
                treeModelListeners.remove(l);
            }
        }

        /**
         * Callback déclenché lorsqu'un noeud est sélectionné dans le {@link #treeView}
         * @param e l'évènement de sélection dans le {@link JTree}
         * @see javax.swing.event.TreeSelectionListener#valueChanged(javax.swing.event.TreeSelectionEven
t)
         * @note doit être réimplémenté dans les classes filles si l'arbre est plus
         * complexe qu'une racine et de figures en dessous.
         */
        @Override
        public void valueChanged(TreeSelectionEvent e)
        {
            JTree tree = (JTree) e.getSource();
            int count = tree.getSelectionCount();
            TreePath[] paths = tree.getSelectionPaths();

            if (¬selfEvent)
            {
                drawing.clearSelection();

                for (int i = 0; i < count; i++)
                {
                    Object[] objPath = paths[i].getPath();
                    int pathSize = paths[i].getPathCount();
                    Object node = objPath[pathSize - 1];
                    if (node ≡ rootElement) // select all figures
                    {
                        drawing.stream().forEach((Figure f ) →
                        {
                            f.setSelected(true);
                        });
                    }
                    if (node instanceof Figure) // Select one figure
                    {
                        Figure figure = (Figure) node;
                        figure.setSelected(true);
                        drawing.stream().forEach((Figure f) →
                        {
                            if (f.equals(figure))
                            {
                                f.setSelected(true);
                            }
                        });
                    }
                }
                drawing.updateSelection();
            }

            selfEvent = false;
        }
}
```

```java
package figures.treemodels;

import java.util.Iterator;
import java.util.List;
import java.util.Vector;

import javax.swing.JTree;
import javax.swing.tree.TreePath;

import figures.Drawing;
import figures.Figure;

/**
 * Figure TreeModel dans lequel les noeuds de niveau 1 sont les figures,
 * Il n'y a pas de noeuds de niveau 2.
 * @author davidroussel
 */
public class FigureTreeModel extends AbstractFigureTreeModel
{
    /**
     * La liste des figure dans l'arbre
     */
    private List<Figure> figures;

    /**
     * Constructeur de l'arbre des types de figures
     * @param drawing le modèle de dessin
     * @param tree le JTree utilisé pour visualiser cet arbre
     */
    public FigureTreeModel(Drawing drawing, JTree tree) throws NullPointerException
    {
        super(drawing, tree, "Figures");
        figures = new Vector<Figure>();
        update(drawing, null); // force Tree build
    }

    /**
     * Mise à jour des figures de l'arbre en les comparant une par une aux
     * figures du modèle de dessin.
     * Permet l'enlever/ajouter les figures de l'arbre en fonction des
     * modifications observées dans les figures du modèle de dessin.
     * @param figures les figures du modèle de dessin
     */
    @Override
    protected synchronized void updateFiguresFromDrawing(List<Figure> figures)
    {
        /*
         * Tant que this.figures n'est pas construit on update pas.
         */
        if (this.figures == null)
        {
            return;
        }

        TreePath parentPath = new TreePath(new Object[] { rootElement });

        /*
         * Comparaison des figures du tree avec les figures du modèle
         * en vue de déterminer
         *  - les noeuds de l'arbre à supprimer : comparaison tree --> model
         *  - les noeuds à ajouter à l'arbre : comparaison model --> tree
         */
        // Comparaison Tree --> Model : noeuds à enlever
        List<Integer> removeChildIndexList = new Vector<Integer>(this.figures.size());
        List<Object> removeNodesList = new Vector<Object>(this.figures.size());
        int nbNodesInitial = this.figures.size();
        int figureIndex = 0;
        for (Iterator<Figure> treeIt = this.figures.iterator(); treeIt.hasNext();)
        {
            Figure figure = treeIt.next();
            if (figures.indexOf(figure) != figureIndex)
            {
                // Cette figure doit être enlevée
                treeIt.remove();
                // Index & Object pour la MAJ Listeners
                removeChildIndexList.add(new Integer(figureIndex));
                removeNodesList.add(figure);
            }
            figureIndex++;
        }

        int nbRemoved = removeChildIndexList.size();
        if (nbRemoved > 0)
        {
            int[] removeChildIndex = new int[nbRemoved];
            for (int i = 0; i < nbRemoved; i++)
            {
                removeChildIndex[i] = removeChildIndexList.get(i).intValue();
            }

            if (nbRemoved < nbNodesInitial)
            {
                fireTreeNodesRemoved(parentPath,
                                     removeChildIndex,
                                     removeNodesList.toArray());
            }
            else
            {
                fireTreeStructureChanged(parentPath);
            }
        }

        // Comparaison Model --> Tree : noeuds à ajouter
        List<Integer> addChildIndexList = new Vector<Integer>(figures.size());
        List<Object> addNodesList = new Vector<Object>(figures.size());
        nbNodesInitial = this.figures.size();
        figureIndex = 0;
        for (Iterator<Figure> drawIt = figures.iterator(); drawIt.hasNext();)
        {
            Figure figure = drawIt.next();
            if (this.figures.indexOf(figure) != figureIndex)
            {
                // Cette figure doit être ajoutée
                this.figures.add(figureIndex, figure);
                // Index & Object pour la MAJ Listeners
                addChildIndexList.add(new Integer(figureIndex));
                addNodesList.add(figure);
            }
            figureIndex++;
        }

        int nbAdded = addChildIndexList.size();
        if (nbAdded > 0)
        {
            int[] addChildIndex = new int[nbAdded];
            for (int i = 0; i < nbAdded; i++)
            {
                addChildIndex[i] = addChildIndexList.get(i).intValue();
            }

            if (nbNodesInitial > 0)
            {
                fireTreeNodesInserted(parentPath,
                                      addChildIndex,
                                      addNodesList.toArray());
            }
            else
            {
                fireTreeStructureChanged(parentPath);
            }
        }
    }

    /**
     * Mise à jour de {@link #selectedFigures} d'après les figures de l'arbre
     * sélectionnées.
     */
    @Override
    protected void updateSelectedFigures()
    {
        if (figures != null)
        {
            // Mise à jour des figures sélectionnées
            for (Iterator<Figure> treeIt = figures.iterator(); treeIt.hasNext();)
            {
                Figure figure = treeIt.next();
                if (figure.isSelected())
                {
                    TreePath selectedPath = new TreePath(new Object[]{
                        rootElement,
                        figure
                    });
                    selectedFigures.add(selectedPath);
                }
            }
        }
    }

    /*
     * (non-Javadoc)
     * @see javax.swing.tree.TreeModel#getChild(java.lang.Object, int)
     */
    @Override
    public Object getChild(Object parent, int index)
    {

        if (parent == rootElement)
        {
            if (figures != null)
            {
```

```
181                 if ((index ≥ 0) ∧ (index < figures.size()))
182                 {
183                     return figures.get(index);
184                 }
185             }
186         }
187 
188         return null;
189     }
190 
191     /*
192      * (non-Javadoc)
193      * @see javax.swing.tree.TreeModel#getChildCount(java.lang.Object)
194      */
195     @Override
196     public int getChildCount(Object parent)
197     {
198         if (parent ≡ rootElement)
199         {
200             if (figures ≠ null)
201             {
202                 return figures.size();
203             }
204         }
205 
206         return 0;
207     }
208 
209     /*
210      * (non-Javadoc)
211      * @see javax.swing.tree.TreeModel#getIndexOfChild(java.lang.Object,
212      * java.lang.Object)
213      */
214     @Override
215     public int getIndexOfChild(Object parent, Object child)
216     {
217         if (parent ≡ rootElement)
218         {
219             if (figures ≠ null)
220             {
221                 return figures.indexOf(child);
222             }
223         }
224 
225         return -1;
226     }
227 
228     /*
229      * (non-Javadoc)
230      * @see javax.swing.tree.TreeModel#isLeaf(java.lang.Object)
231      */
232     @Override
233     public boolean isLeaf(Object node)
234     {
235         if (node ≡ rootElement)
236         {
237             return false;
238         }
239 
240         return true;
241     }
242 
243     /*
244      * (non-Javadoc)
245      * @see java.lang.Object#toString()
246      */
247     @Override
248     public String toString()
249     {
250         StringBuilder sb = new StringBuilder();
251 
252         sb.append(rootElement + "\n");
253 
254         if (figures ≠ null)
255         {
256             for (Figure figure : figures)
257             {
258                 sb.append("+--").append(figure.toString()).append('\n');
259             }
260         }
261 
262         return sb.toString();
263     }
264 }
```



```
1  package figures.treemodels;
2  
3  import java.util.Iterator;
4  import java.util.List;
5  import java.util.Map;
6  import java.util.Set;
7  import java.util.TreeMap;
8  import java.util.Vector;
9  import java.util.concurrent.ConcurrentMap;
10 import java.util.concurrent.ConcurrentSkipListMap;
11 
12 import javax.swing.JTree;
13 import javax.swing.event.TreeSelectionEvent;
14 import javax.swing.tree.TreePath;
15 
16 import figures.Drawing;
17 import figures.Figure;
18 import filters.FigureFilter;
19 
20 /**
21  * Figure TreeModel dans lequel les noeuds de niveau 1 sont une caractéristique
22  * de figures et les noeuds de niveau 2 les figures elles mêmes.
23  * Exemple :
24  * Titre de l'arbre
25  * + Type 1
26  * | + Figure de Type 1 1
27  * | + Figure de Type 1 2
28  * + Type 2
29  * | + Figure de Type 2 1
30  * + Type 3
31  *   + Figure de Type 3 1
32  * @author davidroussel
33  */
34 public abstract class AbstractTypedFigureTreeModel<E> extends AbstractFigureTreeModel
35 {
36     /**
37      * Le Dictionnaire des figures
38      * - Les clefs sont une caractéristique des figures
39      * - Les valeurs des listes de figures correspondant à cette caractéristique
40      * Cette map doit être concurrente ET triée :
41      *  - Concurrente car la méthode {@link #updateFiguresFromDrawing(List)}
42      *  risque d'être appellée de manière récursive à chaque fireXXXEvent
43      *  - Et triée de manière à ce que les clés restent toujours dans le même
44      *  ordre
45      */
46     protected ConcurrentMap<E, List<Figure>> map;
47 
48     /**
49      * L'instance de la classe {@link Class} correspondant aux éléments de
50      * type E de manière à pouvoir comparer les types en utilisant cet attribut
51      */
52     protected Class<E> elementType;
53 
54     /**
55      * Constructeur de l'arbre des types de figures
56      * @param drawing le modèle de dessin
57      * @param tree le JTree utilisé pour visualiser cet arbre
58      * @param title le nom de la racine de cet arbre
59      */
60     public AbstractTypedFigureTreeModel(Class<E> elementType,
61                                         Drawing drawing,
62                                         JTree tree,
63                                         String title)
64         throws NullPointerException
65     {
66         super(drawing, tree, title);
67         map = new ConcurrentSkipListMap<E, List<Figure>>(); // Triée & concurrente
68 
69         if (elementType ≠ null)
70         {
71             this.elementType = elementType;
72         }
73         else
74         {
75             throw new NullPointerException("AbstractTypeFigureTreeModel : null element type");
76         }
77 
78         update(drawing, null); // force Tree build
79     }
80 
81     /**
82      * Nettoyage avant destruction
83      */
84     @Override
85     protected void finalize() throws Throwable
86     {
87         if (map ≠ null)
88         {
89             Set<E> keySet = map.keySet();
90             for (Iterator<E> keyIt = keySet.iterator(); keyIt.hasNext();)
```

```
91              {
92                  List<Figure> keyFigures = map.get(keyIt.next());
93                  keyFigures.clear();
94              }
95              map.clear();
96          }
97      }
98
99      /**
100      * Récupère la valeur de type E d'une figure utilisée pour les noeuds
101      * de niveau 1 de l'arbre
102      * @param f la figure à interroger
103      * @return la valeur de type E contenue dans cette figure en utilisant
104      * l'accesseur adéquat.
105      */
106     public abstract E getValueFrom(Figure f);
107
108     /**
109      * Obtention d'un filtre filtrant les figures possédant la même
110      * caractéristique de type E que la figure f
111      * @param l'élément de type E à utiliser pour le filtre
112      * @return le filtre correspondant à la caractéristique de type E de la
113      * figure f
114      */
115     public abstract FigureFilter<E> getFilter(E element);
116
117     /**
118      * Mise à jour des figures de l'arbre en les comparant une par une aux
119      * figures du modèle de dessin.
120      * Permet l'enlever/ajouter les figures de l'arbre en fonction des
121      * modifications observées dans les figures du modèle de dessin.
122      * @param figures les figures du modèle de dessin
123      */
124     @Override
125     protected void updateFiguresFromDrawing(List<Figure> figures)
126     {
127         /*
128          * Tant que map n'est pas construit on update pas.
129          */
130         if (map == null)
131         {
132             return;
133         }
134
135         /*
136          * Construction d'une map du même type que celle utilisée
137          * dans ce treemodel avec les figures du modèle passées en argument
138          */
139         Map<E, List<Figure>> dmap = new TreeMap<E, List<Figure>>();
140         for (Iterator<Figure> drawIt = figures.iterator(); drawIt.hasNext();)
141         {
142             Figure figure = drawIt.next();
143             E type = getValueFrom(figure);
144             List<Figure> keyFigure = dmap.get(type);
145             if (keyFigure == null)
146             {
147                 dmap.put(type, new Vector<Figure>());
148                 keyFigure = dmap.get(type);
149             }
150             keyFigure.add(figure);
151         }
152
153         TreePath rootPath = new TreePath(new Object[] { rootElement });
154
155         /*
156          * Comparaison des figures du tree avec les figures du modèle
157          * en vue de déterminer
158          *  - les noeuds de l'arbre à supprimer
159          *  - les noeuds à ajouter à l'arbre
160          */
161         // -------------------------------------------------------------------
162         // Comparaison Tree --> Draw : Types, en vue d'enlever des types
163         //-------------------------------------------------------------------
164         List<Integer> removeNodes1IndexList = new Vector<Integer>();
165         List<Object> removeNodes1ObjectList = new Vector<Object>();
166         Set<E> treeKeySet = map.keySet();
167         synchronized (map)
168         {
169             int nbNodesInitialBeforeRemove1 = map.size();
170             int typeIndex = 0;
171             for(Iterator<E> treeKeyIt = treeKeySet.iterator(); treeKeyIt.hasNext(); )
172             {
173                 E treeType = treeKeyIt.next();
174                 if (¬dmap.containsKey(treeType))
175                 {
176                     // Retrait de ce type de la map
177                     map.remove(treeType);
178                     // Index & Object pour la MAJ Listeners
179                     removeNodes1IndexList.add(new Integer(typeIndex));
180                     removeNodes1ObjectList.add(treeType);
```



```
181                 }
182                 typeIndex++;
183             }
184
185             // Notification noeuds 1 supprimés
186             int nbNodes1Removed = removeNodes1IndexList.size();
187             if(nbNodes1Removed > 0)
188             {
189                 int [] removeNodes1Index = new int[nbNodes1Removed];
190                 for (int i = 0; i < nbNodes1Removed; i++)
191                 {
192                     removeNodes1Index[i] = removeNodes1IndexList.get(i).intValue();
193                 }
194                 if (nbNodes1Removed < nbNodesInitialBeforeRemove1)
195                 {
196                     fireTreeNodesRemoved(rootPath,
197                                          removeNodes1Index,
198                                          removeNodes1ObjectList.toArray());
199                 }
200                 else
201                 {
202                     fireTreeStructureChanged(rootPath);
203                 }
204             }
205         }
206
207         //-------------------------------------------------------------------
208         // Comparaison Tree --> Draw : figures, en vue de retirer des figures
209         //-------------------------------------------------------------------
210         treeKeySet = map.keySet();
211         for (Iterator<E> treeKeyIt = treeKeySet.iterator(); treeKeyIt.hasNext();)
212         {
213             E type = treeKeyIt.next();
214             List<Figure> treeKeyFigures = map.get(type);
215             List<Figure> drawKeyFigures = dmap.get(type);
216             List<Integer> removeNodes2IndexList = new Vector<Integer>();
217             List<Object> removeNodes2ObjectList = new Vector<Object>();
218
219             synchronized (map)
220             {
221                 int nbNodesInitialBeforeRemove2 = treeKeyFigures.size();
222                 int figureIndex = 0;
223                 for (Iterator<Figure> treeIt = treeKeyFigures.iterator(); treeIt.hasNext();)
224                 {
225                     Figure figure = treeIt.next();
226                     if (drawKeyFigures.indexOf(figure) ≠ figureIndex)
227                     {
228                         // Cette figure doit être enlevée
229                         treeIt.remove();
230                         // Index & Object pour la MAJ Listeners
231                         removeNodes2IndexList.add(new Integer(figureIndex));
232                         removeNodes2ObjectList.add(figure);
233                     }
234                     figureIndex++;
235                 }
236
237                 int nbRemoved = removeNodes2IndexList.size();
238                 if (nbRemoved > 0)
239                 {
240                     TreePath parentPath = new TreePath(new Object[]{rootElement, type});
241                     int[] removeNodes2Index = new int[nbRemoved];
242                     for (int i = 0; i < nbRemoved; i++)
243                     {
244                         removeNodes2Index[i] = removeNodes2IndexList.get(i).intValue();
245                     }
246
247                     if (nbRemoved < nbNodesInitialBeforeRemove2)
248                     {
249                         fireTreeNodesRemoved(parentPath,
250                                              removeNodes2Index,
251                                              removeNodes2ObjectList.toArray());
252                     }
253                     else
254                     {
255                         fireTreeStructureChanged(parentPath);
256                     }
257                 }
258             }
259         }
260
261         //-------------------------------------------------------------------
262         // Comparaison Draw --> Tree : Types en vue d'ajouter des types
263         //-------------------------------------------------------------------
264         Set<E> drawKeySet = dmap.keySet();
265         List<Integer> addNodes1IndexList = new Vector<Integer>();
266         List<Object> addNodes1ObjectList = new Vector<Object>();
267         synchronized (map)
268         {
269             int nbNodesInitialBeforeAdd1 = map.size();
270             int typeIndex = 0;
```

```
271             for (Iterator<E> drawKeyIt = drawKeySet.iterator(); drawKeyIt.hasNext(); )
272             {
273                 E drawType = drawKeyIt.next();
274                 if (¬map.containsKey(drawType))
275                 {
276                     // Ajout de ce type à map
277                     map.put(drawType, new Vector<Figure>());
278                     // Index & Object pour la MAJ Listeners
279                     addNodes1IndexList.add(new Integer(typeIndex));
280                     addNodes1ObjectList.add(drawType);
281                 }
282                 typeIndex++;
283             }
284 
285             // Notification noeuds 1 ajoutés
286             int nbNodes1Added = addNodes1IndexList.size();
287             if(nbNodes1Added > 0)
288             {
289                 int [] addNodes1Index = new int[nbNodes1Added];
290                 for (int i = 0; i < nbNodes1Added; i++)
291                 {
292                     addNodes1Index[i] = addNodes1IndexList.get(i).intValue();
293                 }
294                 if (nbNodesInitialBeforeAdd1 > 0)
295                 {
296                     fireTreeNodesInserted(rootPath,
297                                           addNodes1Index,
298                                           addNodes1ObjectList.toArray());
299                 }
300                 else
301                 {
302                     fireTreeStructureChanged(rootPath);
303                 }
304             }
305         }
306 
307         //-------------------------------------------------------------------
308         // Comparaison Draw --> Tree : Figures, en vue d'ajouter des figures
309         //-------------------------------------------------------------------
310         for (Iterator<E> drawKeyIt = drawKeySet.iterator(); drawKeyIt.hasNext();)
311         {
312             E type = drawKeyIt.next();
313             List<Figure> drawKeyFigures = dmap.get(type);
314             List<Figure> treeKeyFigures = map.get(type);
315             List<Integer> addNodes2IndexList = new Vector<Integer>();
316             List<Object> addNodes2ObjectList = new Vector<Object>();
317             synchronized (map)
318             {
319                 int nbNodesInitialBeforeAdd2 = treeKeyFigures.size();
320                 int figureIndex = 0;
321                 for (Iterator<Figure> drawIt = drawKeyFigures.iterator(); drawIt.hasNext();)
322                 {
323                     Figure figure = drawIt.next();
324                     if (treeKeyFigures.indexOf(figure) ≠ figureIndex)
325                     {
326                         // Cette figure doit être insérée
327                         treeKeyFigures.add(figureIndex, figure);
328                         // Index & Object pour la MAJ Listeners
329                         addNodes2IndexList.add(new Integer(figureIndex));
330                         addNodes2ObjectList.add(figure);
331                     }
332                     figureIndex++;
333                 }
334 
335                 int nbAdded = addNodes2IndexList.size();
336                 if (nbAdded > 0)
337                 {
338                     TreePath parentPath = new TreePath(new Object[]{rootElement, type});
339                     int[] addNodes2Index = new int[nbAdded];
340                     for (int i = 0; i < nbAdded; i++)
341                     {
342                         addNodes2Index[i] = addNodes2IndexList.get(i).intValue();
343                     }
344 
345                     if (nbNodesInitialBeforeAdd2 > 0)
346                     {
347                         fireTreeNodesInserted(parentPath,
348                                               addNodes2Index,
349                                               addNodes2ObjectList.toArray());
350                     }
351                     else
352                     {
353                         fireTreeStructureChanged(parentPath);
354                     }
355                 }
356             }
357         }
358     }
359 
360     /**
```



```
361      * Mise à jour de {@link #selectedFigures} d'après les figures de l'arbre
362      * sélectionnées.
363      */
364     @Override
365     protected void updateSelectedFigures()
366     {
367         if (map ≠ null)
368         {
369             Set<E> keySet = map.keySet();
370             for (Iterator<E> keyIt = keySet.iterator(); keyIt.hasNext();)
371             {
372                 E type = keyIt.next();
373                 List<Figure> keyFigures = map.get(type);
374                 for (Iterator<Figure> figIt = keyFigures.iterator(); figIt.hasNext();)
375                 {
376                     Figure figure = figIt.next();
377                     if (figure.isSelected())
378                     {
379                         TreePath selectedPath = new TreePath(new Object[]{
380                             rootElement,
381                             type,
382                             figure
383                         });
384                         selectedFigures.add(selectedPath);
385                     }
386                 }
387             }
388         }
389     }
390 
391     /*
392      * (non-Javadoc)
393      * @see javax.swing.tree.TreeModel#getChild(java.lang.Object, int)
394      */
395     @Override
396     public Object getChild(Object parent, int index)
397     {
398         if (map ≠ null)
399         {
400             if (parent ≡ rootElement)
401             {
402                 if ((index ≥ 0) ∧ (index < map.size()))
403                 {
404                     Set<E> keySet = map.keySet();
405                     int count = 0;
406                     E currentKey = null;
407                     for (Iterator<E> keyIt = keySet.iterator();
408                         keyIt.hasNext() ∧ (count ≤ index); count++)
409                     {
410                         currentKey = keyIt.next();
411                     }
412 
413                     return currentKey;
414                 }
415             }
416             else if (elementType.isInstance(parent)) // (parent instanceof E)
417             {
418                 @SuppressWarnings("unchecked")
419                 E type = (E) parent;
420                 List<Figure> keyFigures = map.get(type);
421 
422                 if (keyFigures ≠ null)
423                 {
424                     if ((index ≥ 0) ∧ (index < keyFigures.size()))
425                     {
426                         return keyFigures.get(index);
427                     }
428                 }
429             }
430         }
431 
432         return null;
433     }
434 
435     /*
436      * (non-Javadoc)
437      * @see javax.swing.tree.TreeModel#getChildCount(java.lang.Object)
438      */
439     @Override
440     public int getChildCount(Object parent)
441     {
442         if (map ≠ null)
443         {
444             if (parent ≡ rootElement)
445             {
446                 return map.size();
447             }
448             else if (elementType.isInstance(parent)) // (parent instanceof E)
449             {
450                 @SuppressWarnings("unchecked")
```

```
451                     E type = (E) parent;
452                     List<Figure> keyFigures = map.get(type);
453                     if (keyFigures ≠ null)
454                     {
455                         return keyFigures.size();
456                     }
457                 }
458             }
459 
460             return 0;
461         }
462 
463         /*
464          * (non-Javadoc)
465          * @see javax.swing.tree.TreeModel#isLeaf(java.lang.Object)
466          */
467         @Override
468         public boolean isLeaf(Object node)
469         {
470             if (node instanceof Figure)
471             {
472                 return true;
473             }
474 
475             return false;
476         }
477 
478         /*
479          * (non-Javadoc)
480          * @see javax.swing.tree.TreeModel#getIndexOfChild(java.lang.Object,
481          * java.lang.Object)
482          */
483         @Override
484         public int getIndexOfChild(Object parent, Object child)
485         {
486             if (map ≠ null)
487             {
488                 if (parent ≡ rootElement)
489                 {
490                     // searching in Types for child
491                     Set<E> keySet = map.keySet();
492                     int index = 0;
493                     for (Iterator<E> keyIt = keySet.iterator(); keyIt.hasNext();)
494                     {
495                         if (keyIt.next().equals(child))
496                         {
497                             return index;
498                         }
499                         index++;
500                     }
501                 }
502                 else if (elementType.isInstance(parent)) // (parent instanceof E)
503                 {
504                     // searching in Typed Figures for child
505                     @SuppressWarnings("unchecked")
506                     E type = (E) parent;
507                     List<Figure> keyFigures = map.get(type);
508                     if (keyFigures ≠ null)
509                     {
510                         return keyFigures.indexOf(child);
511                     }
512                 }
513             }
514 
515             return -1;
516         }
517 
518         /**
519          * Callback déclenché lorsqu'un noeud est sélectionné dans le
520          * {@link #treeView}
521          * @param e l'évènement de sélection dans le {@link JTree}
522          * @see javax.swing.event.TreeSelectionListener#valueChanged(javax.swing.event.TreeSelectionEven
    t)
523          */
524         @Override
525         public void valueChanged(TreeSelectionEvent e)
526         {
527             JTree tree = (JTree) e.getSource();
528             int count = tree.getSelectionCount();
529             TreePath[] paths = tree.getSelectionPaths();
530 
531             if (¬selfEvent)
532             {
533                 drawing.clearSelection();
534 
535                 for (int i = 0; i < count; i++)
536                 {
537                     Object[] objPath = paths[i].getPath();
538                     int pathSize = paths[i].getPathCount();
539                     Object node = objPath[pathSize - 1];
```



```
540                 if (node ≡ rootElement) // select all figures
541                 {
542                     drawing.stream().forEach((Figure f) → {
543                         f.setSelected(true);
544                     });
545                 }
546                 if (elementType.isInstance(node)) // select all figures of this type
547                 {
548                     @SuppressWarnings("unchecked")
549                     E type = (E) node;
550                     drawing.stream()
551                             .filter(getFilter(type))
552                             .forEach((Figure f) →
553                             {
554                                 f.setSelected(true);
555                             });
556                 }
557                 if (node instanceof Figure) // Select one figure
558                 {
559                     Figure figure = (Figure) node;
560                     // figure.setSelected(true);
561                     drawing.stream().forEach((Figure f) →
562                     {
563                         if (f.equals(figure))
564                         {
565                             f.setSelected(true);
566                         }
567                     });
568                 }
569             }
570 
571             drawing.updateSelection();
572         }
573 
574         selfEvent = false;
575     }
576 
577     /*
578      * (non-Javadoc)
579      * @see java.lang.Object#toString()
580      */
581     @Override
582     public String toString()
583     {
584         StringBuilder sb = new StringBuilder();
585 
586         sb.append(rootElement).append("\n");
587 
588         if (map ≠ null)
589         {
590             Set<E> keySet = map.keySet();
591             for (Iterator<E> keyIt = keySet.iterator(); keyIt.hasNext();)
592             {
593                 E type = keyIt.next();
594                 sb.append("+--").append(type.toString()).append('s')
595                     .append("\n");
596 
597                 List<Figure> keyFigures = map.get(type);
598                 for (Iterator<Figure> figureIt =
599                     keyFigures.iterator(); figureIt.hasNext();)
600                 {
601                     sb.append("  +--").append(figureIt.next().toString())
602                         .append("\n");
603                 }
604             }
605         }
606 
607         return sb.toString();
608     }
609 }
```

avr 21, 17 11:51 **package−info.java** Page 1/1

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

avr 21, 17 11:51 **FigureFilter.java** Page 1/2

```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 abstract 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 f correspond à l'élément contenu
     * dans ce prédicat (par exemple figure.getType() == element pour filtrer
     * les types de figures)
     * @see java.util.function.Predicate#test(java.lang.Object)
     */
    @Override
    public abstract boolean test(Figure f);

    /**
     * Caomparaison avec un autr objet
     * @param obj l'objet à comparer
     * @erturn 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;
        }

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

```
91                  else
92                  {
93                      if ((element ≠ null) ∨ (ff.element ≠ null))
94                      {
95                          return false;
96                      }
97                      else
98                      {
99                          return true;
100                     }
101                 }
102             }
103         }
104
105         return false;
106     }
107
108     /**
109      * Chaîne de caractères représentant le filtre
110      * @return une chaine de caractère représentantn le filtre
111      */
112     @Override
113     public String toString()
114     {
115         return new String(getClass().getSimpleName() + "<"
116             + (element ≠ null ? element.getClass().getSimpleName() : "null")
117             + ">(" + (element ≠ null ? element.toString() : "") + ")");
118     }
119 }
```



```
1   package filters;
2
3   import java.util.Collection;
4   import java.util.Iterator;
5   import java.util.Vector;
6   import java.util.function.Predicate;
7
8   import figures.Figure;
9
10  /**
11   * Collection de filtres
12   * @author davidroussel
13   */
14  public class FigureFilters<T> implements Collection<FigureFilter<T>>, Predicate<Figure>
15  {
16      /**
17       * Vecteur de filtres
18       */
19      private Vector<FigureFilter<T>> filters;
20
21      /**
22       * Constructeur par défaut
23       */
24      public FigureFilters()
25      {
26          filters = new Vector<FigureFilter<T>>();
27      }
28
29      /**
30       * Test du prédicat
31       * @param f la figure à tester
32       * @return true si l'un au moins des prédicats de la collection est vrai,
33       * false sinon
34       * @see filters.FigureFilter#test(figures.Figure)
35       */
36      @Override
37      public boolean test(Figure f)
38      {
39          boolean result = false;
40
41          for (FigureFilter<T> ff : this)
42          {
43              boolean thisResult = ff.test(f);
44              // System.out.println(ff + (thisResult ? "passed": "denied"));
45              result |= thisResult;
46          }
47
48          // System.out.println(this +  (result ? "passed": "denied"));
49
50          return result;
51      }
52
53      /**
54       * Taille de la collection
55       * @return la taille de la collection
56       */
57      @Override
58      public int size()
59      {
60          return filters.size();
61      }
62
63      /**
64       * Teste si la collection est vide
65       * @return true si la collection est vide
66       */
67      @Override
68      public boolean isEmpty()
69      {
70          return filters.isEmpty();
71      }
72
73      /**
74       * Test de contenu d'un objet dans la collection de filtres
75       * @param o l'objet recherché dans la collection de filtres
76       * @return true si l'objet est contenu dans la collection de filtres
77       */
78      @Override
79      public boolean contains(Object o)
80      {
81          return filters.contains(o);
82      }
83
84      /**
85       * Itérateur de la collection de {@link FigureFilter}
86       * @return l'itérateur sur les filtres de la collection
87       */
88      @Override
89      public Iterator<FigureFilter<T>> iterator()
90      {
```

```
91          return filters.iterator();
92      }
93
94      /**
95       * Conversion en tableau d'objets
96       * @return un tableau d'objets contenant les éléments de la collection
97       */
98      @Override
99      public Object[] toArray()
100     {
101         return filters.toArray();
102     }
103
104     /**
105      * Conversion en tableau générique
106      * @param a un tableau générique spécimen
107      * @return un tableau générique contenant les éléments de la collection
108      */
109     @Override
110     @SuppressWarnings("hiding")
111     public <T> T[] toArray(T[] a)
112     {
113         return filters.toArray(a);
114     }
115
116     /**
117      * Ajout d'un nouveau filtre à la collection uniquement si celle ci ne
118      * contient pas déjà ce filtre
119      * @param filter le filtre à ajouter
120      * @return true si le filtre à ajouté n'était pas déjà présent dans la
121      * collection et qu'il a été ajouté
122      */
123     @Override
124     public boolean add(FigureFilter<T> filter)
125     {
126         if (¬contains(filter))
127         {
128             return filters.add(filter);
129         }
130         else
131         {
132             return false;
133         }
134     }
135
136     /**
137      * Retrait d'un objet de la collection
138      * @param o l'objet à retirer de la collection
139      * @return true si l'objet a été retiré de la collection
140      */
141     @Override
142     public boolean remove(Object o)
143     {
144         return filters.remove(o);
145     }
146
147     /**
148      * Test si une collection est entièrement contenue dans la collection
149      * @param c la collection à tester
150      * @return true si la collection c est entièrement contenue dans la
151      * collection
152      */
153     @Override
154     public boolean containsAll(Collection<?> c)
155     {
156         return filters.containsAll(c);
157     }
158
159     /**
160      * Ajout d'une collection de {@link FigureFilters} à la collection courante
161      * @param c la collection de {@link FigureFilter} à ajouter
162      * @return true si au moins un élément de la collection c a été ajouté
163      * à la collection courante
164      */
165     @Override
166     public boolean addAll(Collection<? extends FigureFilter<T>> c)
167     {
168         boolean added = false;
169         for (FigureFilter<T> ff : c)
170         {
171             if (¬contains(ff))
172             {
173                 added |= add(ff);
174             }
175         }
176
177         return added;
178     }
179
180     /**
```



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

```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)
    {
        Integer key = Integer.valueOf(hash);
        if (map.containsKey(key))
        {
            return map.get(key);
        }

        return null;
    }

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

import javax.swing.ImageIcon;

/**
 * Classe contenant une FlyweigtFactory 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)
            {
                URL url = IconFactory.class.getResource(ImageBase + name + ImageType);
                if (url ≠ null)
                {
                    icon = new ImageIcon(url);
                    if (icon ≠ null ∧
                        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 contenant les différents widgets (éléments graphiques)
 */
package widgets;
```



```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
```

```
91          * de couleus
92          * @return
93          */
94      public static Paint getPaint(Component component,
95                                   String title,
96                                   Color initialColor)
97      {
98          if (component ≠ null)
99          {
100             Color color = JColorChooser.showDialog(component, title, initialColor);
101             if (color ≠ null)
102             {
103                 return paintFactory.get(color);
104             }
105         }
106
107         return null;
108     }
109 }
```



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

```java
package utils;

import java.awt.geom.AffineTransform;
import java.awt.geom.Point2D;

/**
 * Vector 2D class relating two points
 * @author davidroussel
 */
public class Vector2D
{
    /**
     * Vector's origin
     */
    protected Point2D start;

    /**
     * Vector's end
     */
    protected Point2D end;

    /**
     * Construtor from 1 point (the origin is supposed to be (0, 0)
     * @param p the end point
     */
    public Vector2D(Point2D p)
    {
        this(null, p);
    }

    /**
     * Constructor from two points
     * @param p1 the start point
     * @param p2 the end point
     */
    public Vector2D(Point2D p1, Point2D p2)
    {
        start = p1;
        end = p2;
    }

    /**
     * Constructeur de copie
     * @param vector le vecteur à copier
     */
    public Vector2D(Vector2D vector)
    {
        start = vector.start;
        end = vector.end;
    }

    /**
     * Start point getter
     * @return the start
     */
    public Point2D getStart()
    {
        if (start == null)
        {
            return new Point2D.Double(0.0, 0.0);
        }
        else
        {
            return start;
        }
    }

    /**
     * Start point setter
     * @param start the start to set
     */
    public void setStart(Point2D start)
    {
        this.start = start;
    }

    /**
     * End Point getter
     * @return the end
     */
    public Point2D getEnd()
    {
        return end;
    }

    /**
     * End point setter
     * @param end the end to set
     */
    public void setEnd(Point2D end)
```



```java
    {
        this.end = end;
    }

    /**
     * Delta X of the vector
     * @return The delta X of the vector
     */
    protected double getX()
    {
        return end.getX() - (start == null ? 0.0 : start.getX());
    }

    /**
     * Delta Y of the vector
     * @return The delta Y of the vector
     */
    protected double getY()
    {
        return end.getY() - (start == null ? 0.0 : start.getY());
    }

    /**
     * Dot product with vector v
     * @param v the vector to compute dot product with
     * @return the value of the dot product
     */
    public double dotProduc(Vector2D v)
    {
        return ((start == null ? 0.0 : start.getX()) * end.getX()) +
               ((start == null ? 0.0 : start.getY()) * end.getY());
    }

    /**
     * Cross product's norm
     * @param v the vector to compute cross product's norm
     * @return the value of the cross product's norm
     */
    public double crossProductNorm(Vector2D v)
    {
        return (getX()*v.getY()) - (v.getX()*getY());
    }

    /**
     * Vector's norm
     * @return the vector's norm
     */
    public double norm()
    {
        return Math.sqrt(dotProduc(this));
    }

    /**
     * Compute normalized vector's
     * @return normalized vector
     */
    public Vector2D normalize()
    {
        double norm = norm();

        return new Vector2D(new Point2D.Double(getX()/ norm, getY() / norm));
    }

    /**
     * Angle between vectors
     * @param v the vector to compute angle with
     * @return the angle between current vector and vector v
     */
    public double angle(Vector2D v)
    {
        Vector2D vn1 = normalize();
        Vector2D vn2 = v.normalize();

        return Math.atan2(vn2.getY(),vn2.getX()) -
               Math.atan2(vn1.getY(),vn1.getX());
    }

    /**
     * The endPoint as in {@link #end} - {@link #start}
     * @return the end point
     */
    public Point2D toPoint2D()
    {
        return new Point2D.Double(end.getX() - start.getX(),
                                  end.getY() - start.getY());
    }

    /**
     * Apply Affine transform to vector centered on {@link #start}
     * @param transform the affine transform to apply
```

```
181      */
182     public void transformEnd(AffineTransform transform)
183     {
184         Point2D pVector = toPoint2D();
185         if (transform ≠ null)
186         {
187             Point2D tPVector = new Point2D.Double();
188             transform.transform(pVector, tPVector);
189 
190             setEnd(new Point2D.Double(start.getX() + tPVector.getX(),
191                                       start.getY() + tPVector.getY()));
192         }
193     }
194 }
```



```
1   package utils;
2   
3   import java.awt.Color;
4   import java.awt.color.ColorSpace;
5   
6   /**
7    * Une Couleur comparable (pour pouvoir être utilisée dans un ensemble ou un
8    * arbre trié)
9    * @author davidroussel
10   */
11  public class CColor extends Color implements Comparable<CColor>
12  {
13  
14      /**
15       * Instance statique particulière pour représenter pas de couleur
16       */
17      public static final CColor NoColor = new CColor(255, 255, 255, 255);
18  
19      /**
20       * Constructeur à partir d'une couleur ordinaire
21       * @param c la couleur à convertir
22       */
23      public CColor(Color c)
24      {
25          super(c.getRed(), c.getGreen(), c.getBlue(), c.getAlpha());
26      }
27  
28      /**
29       * Constructeur de copie
30       * @param c la couleur comparable à copier
31       */
32      public CColor(CColor c)
33      {
34          this(c.getRed(), c.getGreen(), c.getBlue(), c.getAlpha());
35      }
36  
37      /**
38       * Couleur à partir d'un entier
39       * @param rgb entier dont on utilise les 24 premiers bits pour fabriquer une
40       * couleur
41       */
42      public CColor(int rgb)
43      {
44          super(rgb);
45      }
46  
47      /**
48       * Constructeur à partir d'un entier
49       * @param rgba entier dont on utilise les 32 bits pour fabriquer une
50       * couleur
51       * @param hasalpha indique s'il faut utiliser les 8 dernier bits comme
52       * bits de transparence
53       */
54      public CColor(int rgba, boolean hasalpha)
55      {
56          super(rgba, hasalpha);
57      }
58  
59      /**
60       * Constructeur à partir des composantes R, G & B
61       * @param r la composante rouge
62       * @param g la composante verte
63       * @param b la compostante bleue
64       */
65      public CColor(int r, int g, int b)
66      {
67          super(r, g, b);
68      }
69  
70      /**
71       * Constructeur à partir des composantes R, G & B
72       * @param r la composante rouge
73       * @param g la composante verte
74       * @param b la compostante bleue
75       */
76      public CColor(float r, float g, float b)
77      {
78          super(r, g, b);
79      }
80  
81      /**
82       * Constructeur à partir de composantes dans un espace de couleur particulier
83       * @param cspace l'espace de couleurs utilisé
84       * @param components les composantes dans cet espace de couleur
85       * @param alpha la transparence
86       */
87      public CColor(ColorSpace cspace, float[] components, float alpha)
88      {
89          super(cspace, components, alpha);
90      }
```

```
91
92      /**
93       * Constructeur à partir des composantes R, G & B et Alpha pour la
94       * transparence
95       * @param r la composante rouge
96       * @param g la composante verte
97       * @param b la composante bleue
98       * @param a la composante alpha
99       */
100     public CColor(int r, int g, int b, int a)
101     {
102         super(r, g, b, a);
103     }
104
105     /**
106      * Constructeur à partir des composantes R, G & B et Alpha pour la
107      * transparence
108      * @param r la composante rouge
109      * @param g la composante verte
110      * @param b la composante bleue
111      * @param a la composante alpha
112      */
113     public CColor(float r, float g, float b, float a)
114     {
115         super(r, g, b, a);
116     }
117
118     @Override
119     public int compareTo(CColor o)
120     {
121         int red1 = getRed();
122         int red2 = o.getRed();
123         if (red1 < red2)
124         {
125             return -1;
126         }
127         else // red >= o.red
128         {
129             if (red1 > red2)
130             {
131                 return 1;
132             }
133             else // red1 == red2
134             {
135                 int green1 = getGreen();
136                 int green2 = o.getGreen();
137                 if (green1 < green2)
138                 {
139                     return -1;
140                 }
141                 else // green1 >= green2
142                 {
143                     if (green1 > green2)
144                     {
145                         return 1;
146                     }
147                     else // green1 == green2
148                     {
149                         int blue1 = getBlue();
150                         int blue2 = o.getBlue();
151                         if (blue1 < blue2)
152                         {
153                             return -1;
154                         }
155                         else // blue1 >= blue2
156                         {
157                             if (blue1 > blue2)
158                             {
159                                 return 1;
160                             }
161                             else // blue1 == blue2
162                             {
163                                 int alpha1 = getAlpha();
164                                 int alpha2 = o.getAlpha();
165                                 if (alpha1 < alpha2)
166                                 {
167                                     return -1;
168                                 }
169                                 else // alpha1 >= alpha2
170                                 {
171                                     if (alpha1 > alpha2)
172                                     {
173                                         return 1;
174                                     }
175                                     else
176                                     {
177                                         return 0;
178                                     }
179                                 }
180                             }
```



```
181                             }
182                         }
183                     }
184                 }
185             }
186         }
187
188         /* (non-Javadoc)
189          * @see java.awt.Color#toString()
190          */
191         @Override
192         public String toString()
193         {
194             if (this == NoColor)
195             {
196                 return new String("No Color");
197             }
198             if (super.equals(Color.BLACK))
199             {
200                 return new String("Black");
201             }
202             if (super.equals(Color.BLUE))
203             {
204                 return new String("Blue");
205             }
206             if (super.equals(Color.CYAN))
207             {
208                 return new String("Cyan");
209             }
210             if (super.equals(Color.DARK_GRAY))
211             {
212                 return new String("Dark Gray");
213             }
214             if (super.equals(Color.GRAY))
215             {
216                 return new String("Gray");
217             }
218             if (super.equals(Color.GREEN))
219             {
220                 return new String("Green");
221             }
222             if (super.equals(Color.LIGHT_GRAY))
223             {
224                 return new String("Light Gray");
225             }
226             if (super.equals(Color.MAGENTA))
227             {
228                 return new String("Magenta");
229             }
230             if (super.equals(Color.ORANGE))
231             {
232                 return new String("Orange");
233             }
234             if (super.equals(Color.PINK))
235             {
236                 return new String("Pink");
237             }
238             if (super.equals(Color.RED))
239             {
240                 return new String("Red");
241             }
242             if (super.equals(Color.WHITE))
243             {
244                 return new String("White");
245             }
246             if (super.equals(Color.YELLOW))
247             {
248                 return new String("Yellow");
249             }
250
251             return new String("(" + String.valueOf(getRed()) +
252                                ","  + String.valueOf(getGreen()) +
253                                ","  + String.valueOf(getBlue()) + ")");
254         }
255     }
```

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



```java
package history;

import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;

/**
 * Classe permettant de gérer les piles de Undo et de Redo de E
 * @param E l'état à sauvegarder dans les piles
 * @author davidroussel
 */
public class HistoryManager<E> extends Prototype<E>>
{
    /**
     * Le nombre maximum d'undo / redo
     */
    private int size;

    /**
     * L'originator dont on doit sauvegarder l'état.
     * Permet de demander à l'originator de générer un memento ou de
     * mettre en place un memento qu'on lui fournit
     */
    private Originator<E> originator;

    /**
     * La pile des Undo
     * @note les {@link Deque} permettent d'empiler/dépiler en tête de liste
     * mais aussi d'accéder au dernier élément pour garder des piles de taille
     * inférieure ou égale à {@link #size}
     */
    private Deque<Memento<E>> undoStack;

    /**
     * La pile de Redo
     */
    private Deque<Memento<E>> redoStack;

    /**
     * Constructeur du manager de Undo/Redo
     */
    public HistoryManager(Originator<E> origin, int size)
    {
        this.size = size;
        originator = origin;
        undoStack = new LinkedList<Memento<E>>();
        redoStack = new LinkedList<Memento<E>>();
    }

    @Override
    protected void finalize() throws Throwable
    {
        undoStack.clear();
        redoStack.clear();
        super.finalize();
    }

    /**
     * Nombre d'éléments accumulés dans la pile de undo
     * @return Le nombre d'éléments accumulés dans la pile de undo
     */
    public int undoSize()
    {
        return undoStack.size();
    }

    /**
     * Nombre d'éléments accumulés dans la pile de redo
     * @return Le nombre d'éléments accumulés dans la pile de redo
     */
    public int redoSize()
    {
        return redoStack.size();
    }

    /**
     * Ajout d'un état dans la pile des undo
     * @param state l'état à ajouter dans la pile des undo
     * @return true si le memento était non null, différent du dernier
     * Memento ajouté à la pile des undo et a été ajouté à la pile des undo
     * @note si le nombre d'états dans la pile des undo dépasse {@link #size}
     * alors le tout premier état empilé est supprimé de la pile
     */
    private boolean pushUndo(Memento<E> state)
    {
        if (state ≠ null)
        {
            /*
             * TODO
             *  - On vérifie que le memento que l'on cherche à ajouter est
```

```
91                *  bien différent du dernier ajouté
92                *  - On ajoute ce memento à la pile des undo
93                *  - Si le nombre de mementos dans la pile dépasse #size alors
94                *  on enlève le premier memento de manière à a garder au maximum
95                *  #size mementos dans la pile
96                */
97           }
98           else
99           {
100              System.err.println("HistoryManager::pushUndo(null)");
101          }
102          return false;
103      }
104
105      /**
106       * Dépilage du dernier état empilé dans la pile des undo
107       * @return l'état qui était en haut de la pile des undo, ou bien null
108       * s'il n'y avait pas d'état en haut de la pile
109       */
110      private Memento<E> popUndo()
111      {
112          Memento<E> state = null;
113
114          /*
115           * TODO dépiler le dernier memento empilé
116           */
117
118          return state;
119      }
120
121      /**
122       * Ajout d'un état dans la pile des redo
123       * @param state l'état à ajouter dans la pile des redo
124       * @return true si le memento était non null, différent du dernier
125       * Memento ajouté à la pile des undo et a été ajouté à la pile des undo
126       * @note si le nombre d'états dans la pile des redo dépasse {@link #size}
127       * alors le tout premier état empilé est supprimé de la pile
128       */
129      private boolean pushRedo(Memento<E> state)
130      {
131          if (state ≠ null)
132          {
133              /*
134               * TODO
135               *  - On vérifie que le memento que l'on cherche à ajouter est
136               *  bien différent du dernier ajouté
137               *  - On ajoute ce memento à la pile des redo
138               *  - Si le nombre de mementos dans la pile dépasse #size alors
139               *  on enlève le premier memento de manière à a garder au maximum
140               *  #size mementos dans la pile
141               */
142          }
143          else
144          {
145              System.err.println("HistoryManager::pushRedo(null)");
146          }
147          return false;
148      }
149
150      /**
151       * Dépilage du dernier état empilé dans la pile des redo et donc réempilage
152       * de cet état dans la pile des undo
153       * @return l'état dépilé de la pile des redo
154       */
155      private Memento<E> popRedo()
156      {
157          Memento<E> state = null;
158
159          /*
160           * TODO dépiler le dernier memento empilé
161           */
162
163          return state;
164      }
165
166      /**
167       * Enregistre un {@link Memento} de l'{@link #originator} pour pouvoir
168       * le restituer par la suite.
169       */
170      public void record()
171      {
172          /*
173           * TODO
174           *  - Demander à l'originator de créer un memento
175           *  - Empiler ce memento dans la pile des undo
176           *  - Effacer la pile des redo
177           */
178      }
179
180      /**
```



```
181       * Restitue le dernier Memento sauvegardé dans la pile des undo
182       * @return le dernier memento sauvegardé dans la pile des undo
183       * ({@link #undoStack}), ou bien null si celle-ci est vide.
184       * @post un {@link Memento} de l'{@link #originator} a été créé au préalable
185       * dans la pile des redo.
186       */
187      public Memento<E> undo()
188      {
189          /*
190           * TODO
191           *  - Dépiler un élément de la pile des undo (s'il y en a un)
192           *  - Tout en sauvegardant l'état courant de l'originator dans la pile
193           *  des redo.
194           */
195
196          return null;
197      }
198
199      /**
200       * Annule le dernier {@link Memento} enregistré dans la pile des undo.
201       * Lorsque l'action n'a pas modifié l'état (par exemple si elle a échoué)
202       */
203      public void cancel()
204      {
205          /*
206           * TODO Annuler le dernier empilage dans la pile des undo
207           */
208      }
209
210      /**
211       * Restitue de dernier Memento sauvegardé dans la pile des redo
212       * @return Le dernier Memento sauvegardé dans la pile des redo
213       * ({@link #redoStack}) ou bien null si celle-ci est vide.
214       * @post un {@link Memento} de l'{@link #originator} a été créé au préalable
215       * dans la pile des undo.
216       */
217      public Memento<E> redo()
218      {
219          /*
220           * TODO
221           *  - Dépiler un élément de la pile des redo (s'il y en a un)
222           *  - Tout en sauvegardant l'état courant de l'originator dans la pile
223           *  des undo
224           */
225          return null;
226      }
227
228      @Override
229      public String toString()
230      {
231          StringBuilder sb = new StringBuilder();
232
233          sb.append(super.toString());
234          sb.append("[" + String.valueOf(size) + "]:\nUndo = {");
235          for (Iterator<Memento<E>> it = undoStack.iterator(); it.hasNext();)
236          {
237              sb.append(it.next());
238              if (it.hasNext())
239              {
240                  sb.append(",");
241              }
242          }
243          sb.append("},\nRedo = {");
244          for (Iterator<Memento<E>> it = redoStack.iterator(); it.hasNext();)
245          {
246              sb.append(it.next());
247              if (it.hasNext())
248              {
249                  sb.append(",");
250              }
251          }
252          sb.append("}");
253          return sb.toString();
254      }
255  }
```

```java
package history;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * Un état constitué d'une liste de d'éléments de type E constituant
 * l'état à sauvegarder dans le Memento.
 * @note les élements doivent dériver de {@link Prototype} pour pouvoir
 * être effectivement clonés (Deep Copy) dans l'état du Memento.
 * @author davidroussel
 */
public class Memento<E> extends Prototype<E>>
{
    /**
     * La liste d'élément de type E qui constitue l'état à sauvegarder
     */
    private List<E> state;

    /**
     * Constructeur par défaut d'un état
     */
    public Memento(List<E> things)
    {
        this.state = new ArrayList<E>();
        for (E elt : things)
        {
            this.state.add(elt.clone());
        }
    }

    /**
     * Accesseur à l'état du memento
     * @return l'état stocké dans le memento
     */
    public List<E> getState()
    {
        return state;
    }

    /* (non-Javadoc)
     * @see java.lang.Object#hashCode()
     */
    @Override
    public int hashCode()
    {
        final int prime = 31;
        int hash = 1;
        for (E elt : state)
        {
            hash += (prime * hash) + (elt ≠ null ? elt.hashCode() : 0);
        }
        return hash;
    }

    /**
     * Comparaison entre deux memento.
     * Permet de vérifier que les memento stockés dans l'History manager
     * ne sont pas identiques
     * @param obj l'objet à comparer
     * @return true si les deux memento sont identiques en terme de contenu
     * @see java.lang.Object#equals(java.lang.Object)
     */
    @Override
    public boolean equals(Object obj)
    {
        if (obj ≡ null)
        {
            return false;
        }

        if (obj ≡ this)
        {
            return true;
        }

        if (obj instanceof Memento<?>)
        {
            Memento<?> as = (Memento<?>) obj;
            if (¬as.state.isEmpty() ∧ ¬state.isEmpty())
            {
                if (state.get(0).getClass() ≡ as.state.get(0).getClass())
                {
                    @SuppressWarnings("unchecked")
                    Memento<E> s = (Memento<E>) obj;
                    Iterator<E> it1 = state.iterator();
                    Iterator<E> it2 = s.state.iterator();

                    for(; it1.hasNext() ∧ it2.hasNext();)
                    {
                        if (¬it1.next().equals(it2.next()))
                        {
                            return false;
                        }
                    }

                    return it1.hasNext() ≡ it2.hasNext();
                }
            }
            else
            {
                if (as.state.isEmpty() ∧ state.isEmpty())
                {
                    return true;
                }
            }
        }

        return false;
    }

    /* (non-Javadoc)
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString()
    {
        StringBuilder sb = new StringBuilder();

        sb.append('[');
        for (Iterator<E> it = state.iterator(); it.hasNext();)
        {
            sb.append(it.next());
            if (it.hasNext())
            {
                sb.append(",");
            }
        }
        sb.append(']');

        return sb.toString();
    }
}
```

```java
package history;

/**
 * Interface pour les classes créant et récupérant des Memento de leur état
 * @author davidroussel
 */
public interface Originator<E extends Prototype<E>>
{
    /**
     * Création d'un Memento
     * @return le Memento contenant l'état de l'Originator
     */
    public abstract Memento<E> createMemento();

    /**
     * Remplacement de l'état courant par celui contenu dans le Memento
     * @param memento le memento contenant l'état à mettre en place
     * @post l'état contenu dans le Memento a remplacé l'état courant,
     * SAUF si le memento est null
     */
    public abstract void setMemento(Memento<E> memento);

}
```



```java
package history;

/**
 * Une interface déclarant un prototype public
 * (contrairement à Object qui possède une méthode clone mais qui est protégée)
 * @author davidroussel
 */
public interface Prototype<E>
{
    /**
     * Création d'une copie (distincte mais égale)
     * @return la copie de l'opjet à cloner
     */
    public E clone();
}
```

```java
package widgets;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
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.InputEvent;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import java.util.ArrayList;
import java.util.EventObject;
import java.util.List;

import javax.swing.AbstractAction;
import javax.swing.AbstractButton;
import javax.swing.Action;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JColorChooser;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSeparator;
import javax.swing.JSpinner;
import javax.swing.JTabbedPane;
import javax.swing.JToolBar;
import javax.swing.KeyStroke;
import javax.swing.SpinnerNumberModel;
import javax.swing.SwingConstants;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

import figures.Drawing;
import figures.Figure;
import figures.enums.FigureType;
import figures.enums.LineType;
import figures.enums.PaintToType;
import figures.listeners.AbstractFigureListener;
import figures.listeners.SelectionFigureListener;
import figures.listeners.creation.AbstractCreationListener;
import figures.listeners.transform.AbstractTransformShapeListener;
import figures.listeners.transform.MoveShapeListener;
import history.HistoryManager;
import utils.IconFactory;
import utils.PaintFactory;
import widgets.enums.OperationMode;

/**
 * Classe de la fenêtre principale de l'éditeur de figures
 * @author davidroussel
 */
@SuppressWarnings("serial")
public class EditorFrame extends JFrame
{
    /**
     * Le nom de l'éditeur
     */
    protected static final String EditorName = "Figure Editor v4.0";

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

    /**
     * Le gestionnaire d'historique pour les Undo/Redo
     */
    protected HistoryManager<Figure> history;

    /**
     * Taille de l'historique
     */
    protected static final int historyLength = 32;

    /**
     * Indique si l'éditeur est en mode Création de figures ou édition
     * de figures (mode initial : création de figures)
```



```java
     */
    protected OperationMode operationMode = OperationMode.CREATION;

    /**
     * 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 listener à mettre en place dans le drawingPanel lorsque l'on
     * est en mode édition de figures pour déplacer les figures sélectionnées
     */
    protected AbstractTransformShapeListener moveListener;

    /**
     * Le listener à mettre en place dans le drawingPanel lorsque l'on
     * est en mode édition de figures pour faire tourner les figures
     * sélectionnées
     */
    protected AbstractTransformShapeListener rotateListener;

    /**
     * Le listener à mettre en place dans le drawingPanel lorsque l'on
     * est en mode édition de figures pour changer l'échelle les figures
     * sélectionnées
     */
    protected AbstractTransformShapeListener scaleListener;

    /**
     * Le listener de sélection des figures à mettre en place lorsque l'on
     * est en mode édition.
     */
    protected AbstractFigureListener selectionListener;

    /**
     * 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", "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;

    /**
     * L'index de l'élément sélectionné par défaut dans les types de
     * trait
     */
    private final static int defaultEdgeTypeIndex = 1; // solid

    /**
     * La largeur de trait par défaut
     */
    private final static int defaultEdgeWidth = 4;

    /**
     * Largeur de trait minimum
     */
    private final static int minEdgeWidth = 1;

    /**
     * Largeur de trait maximum
     */
    private final static int maxEdgeWidth = 30;

    /**
     * l'incrément entre deux largeurs de trait
     */
    private final static int stepEdgeWidth = 1;

    /**
     * Action déclenchée lorsque l'on clique sur le bouton quit ou sur l'item
     * de menu quit
     */
    private final Action quitAction = new QuitAction();

    /**
     * Action déclenchée lorsque l'on clique sur le bouton undo ou sur l'item
     * de menu undo
     */
    private final Action undoAction = new UndoAction();

    /**
     * Action réalisée lorsque l'on souhaite refaire une action qui vient
     * d'être annulée
     */
    private final Action redoAction = new RedoAction();

    /**
     * Action déclenchée lorsque l'on clique sur le bouton clear ou sur l'item
     * de menu clear
     */
    private final Action clearAction = new ClearAction();

    /**
     * Action déclenchée lorsque l'on clique sur le bouton about ou sur l'item
     * de menu about
     */
    private final Action aboutAction = new AboutAction();

    /**
     * Action déclenchée lorsque l'on sélectionne de mode édition des figures
     */
    private final Action toggleCreateEditAction = new ToggleCreateEditAction();

    /**
     * Action déclenchée pour mettre filter ou non les figures
     */
```



```
    private final Action filterAction = new FilterAction();

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

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

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

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

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

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

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

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

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

    /**
     * Action déclenchée pour mettre filter ou non les figures suivant
     * la couleur de replissage courante
     */
    private final Action fillColorFilterAction = new FillColorFilterAction();

    /**
     * Action déclenchée pour mettre filter ou non les figures suivant
     * la couleur de trait courante
     */
    private final Action edgeColorFilterAction = new EdgeColorFilterAction();

    /**
     * Action réalisée pour détruire les figures sélectionnées
     */
    private final Action deleteAction = new DeleteAction();

    /**
     * Action réalisée pour monter les figures sélectionnées en tête de liste
     * des figures
     */
    private final Action moveUpAction = new MoveUpAction();

    /**
     * Action réalisée pour descendre les figures sélectionnées en fin de liste
```

```
361      * des figures
362      */
363     private final Action moveDownAction = new MoveDownAction();
364
365     /**
366      * Action réalisée pour appliquer le style courant (couleur de remplissage,
367      * couleur de trait et style de trait) aux figures sélectionnées
368      */
369     private final Action styleAction = new StyleAction();
370
371     /**
372      * Constructeur de la fenêtre de l'éditeur.
373      * Construit les widgets et assigne les actions et autres listeners
374      * aux widgets
375      * @throws HeadlessException
376      */
377     public EditorFrame() throws HeadlessException
378     {
379         drawingModel = new Drawing();
380         history = new HistoryManager<Figure>(drawingModel, historyLength);
381         operationMode = OperationMode.CREATION;
382
383         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
384         boolean isMacOS = System.getProperty("os.name").startsWith("Mac OS");
385
386         /*
387          * Construire l'interface graphique en utilisant WindowBuilder:
388          * Menu Contextuel -> Open With -> WindowBuilder Editor puis
389          * aller dans l'onglet Design
390          */
391         setPreferredSize(new Dimension(650, 450));
392         creationListener = null;
393
394         setTitle(EditorName);
395         if (¬isMacOS)
396         {
397             setIconImage(Toolkit.getDefaultToolkit()
398                 .getImage(EditorFrame.class.getResource("/images/Logo.png")));
399         }
400
401         // ---------------------------------------------------------------
402         // Toolbar en haut
403         // ---------------------------------------------------------------
404         JToolBar toolBar = new JToolBar();
405         toolBar.setFloatable(false);
406         getContentPane().add(toolBar, BorderLayout.NORTH);
407
408         JButton btnCancel = new JButton("Undo");
409         btnCancel.setAction(undoAction);
410         toolBar.add(btnCancel);
411
412         JButton btnRedo = new JButton("Redo");
413         btnRedo.setAction(redoAction);
414         toolBar.add(btnRedo);
415
416         Component toolBoxSpringer = Box.createHorizontalGlue();
417         toolBar.add(toolBoxSpringer);
418
419         JButton btnAbout = new JButton("About");
420         btnAbout.setAction(aboutAction);
421         toolBar.add(btnAbout);
422
423         JButton btnClose = new JButton("Close");
424         btnClose.setAction(quitAction);
425         toolBar.add(btnClose);
426
427         // ---------------------------------------------------------------
428         // Barre d'état en bas
429         // ---------------------------------------------------------------
430         JPanel bottomPanel = new JPanel();
431         getContentPane().add(bottomPanel, BorderLayout.SOUTH);
432         bottomPanel.setLayout(new BoxLayout(bottomPanel, BoxLayout.X_AXIS));
433
434         infoLabel = new JLabel(AbstractFigureListener.defaultTip);
435         bottomPanel.add(infoLabel);
436
437         Component horizontalGlue = Box.createHorizontalGlue();
438         bottomPanel.add(horizontalGlue);
439
440         JLabel coordsLabel = new JLabel(DrawingPanel.defaultCoordString);
441         bottomPanel.add(coordsLabel);
442
443         // ---------------------------------------------------------------
444         // Panneau de contrôle à gauche
445         // ---------------------------------------------------------------
446         JPanel leftPanel = new JPanel();
447         leftPanel.setPreferredSize(new Dimension(220, 10));
448         leftPanel.setAlignmentY(Component.TOP_ALIGNMENT);
449         getContentPane().add(leftPanel, BorderLayout.WEST);
450
```



```
451         JLabeledComboBox figureTypeCombobox = new JLabeledComboBox("Shape",
452                                                                    FigureType
453                                                                        .stringValues(),
454                                                                    defaultFigureTypeIndex,
455                                                                    (ItemListener) null);
456         figureTypeCombobox.setAlignmentX(Component.CENTER_ALIGNMENT);
457         figureTypeCombobox.setPreferredSize(new Dimension(80, 32));
458         leftPanel.setLayout(new BoxLayout(leftPanel, BoxLayout.Y_AXIS));
459         leftPanel.add(figureTypeCombobox);
460
461         JPanel edgeWidthPanel = new JPanel();
462         edgeWidthPanel.setPreferredSize(new Dimension(80, 32));
463         leftPanel.add(edgeWidthPanel);
464         edgeWidthPanel
465             .setLayout(new BoxLayout(edgeWidthPanel, BoxLayout.X_AXIS));
466         SpinnerNumberModel snm =
467             new SpinnerNumberModel(defaultEdgeWidth,
468                                    minEdgeWidth,
469                                    maxEdgeWidth,
470                                    stepEdgeWidth);
471
472         JTabbedPane tabbedPane = new JTabbedPane(SwingConstants.TOP);
473         tabbedPane.setTabLayoutPolicy(JTabbedPane.SCROLL_TAB_LAYOUT);
474         tabbedPane.setAlignmentY(Component.TOP_ALIGNMENT);
475         leftPanel.add(tabbedPane);
476
477         InfoPanel infoPanel = new InfoPanel();
478         infoPanel.setAlignmentY(Component.TOP_ALIGNMENT);
479         tabbedPane.addTab("Info", IconFactory.getIcon("Details_small"), infoPanel, "Selected Figure");
480
481         // ---------------------------------------------------------------
482         // Zone de dessin
483         // ---------------------------------------------------------------
484         JScrollPane scrollPane = new JScrollPane();
485         getContentPane().add(scrollPane, BorderLayout.CENTER);
486
487         drawingPanel = new DrawingPanel(drawingModel, coordsLabel, infoPanel);
488         scrollPane.setViewportView(drawingPanel);
489
490         // ---------------------------------------------------------------
491         // Barre de menus
492         // ---------------------------------------------------------------
493         JMenuBar menuBar = new JMenuBar();
494         setJMenuBar(menuBar);
495
496         JMenu mnFile = new JMenu("Drawing");
497         menuBar.add(mnFile);
498
499         JMenuItem mntmCancel = new JMenuItem("Cancel");
500         mntmCancel.setAction(undoAction);
501         mnFile.add(mntmCancel);
502
503         JMenuItem mntmRedo = new JMenuItem("Redo");
504         mntmRedo.setAction(redoAction);
505         mnFile.add(mntmRedo);
506
507         JMenuItem mntmClear = new JMenuItem("Clear");
508         mntmClear.setAction(clearAction);
509         mnFile.add(mntmClear);
510
511         JMenu mnEdition = new JMenu("Edition");
512         menuBar.add(mnEdition);
513
514         JMenu mnFilter = new JMenu("Filter");
515         menuBar.add(mnFilter);
516
517         JCheckBoxMenuItem chckbxmntmFiltering =
518             new JCheckBoxMenuItem("Filtering");
519         chckbxmntmFiltering.setAction(filterAction);
520         mnFilter.add(chckbxmntmFiltering);
521
522         JMenu mnFigures = new JMenu("Figures");
523         mnFilter.add(mnFigures);
524
525         JMenu mnColors = new JMenu("Colors");
526         mnFilter.add(mnColors);
527
528         JMenu mnStrokes = new JMenu("Strokes");
529         mnFilter.add(mnStrokes);
530
531         JSeparator separator = new JSeparator();
532         mnFile.add(separator);
533
534         JMenuItem mntmQuit = new JMenuItem("Quit");
535         mntmQuit.setAction(quitAction);
536         mnFile.add(mntmQuit);
537
538         JMenu mnHelp = new JMenu("Help");
539         menuBar.add(mnHelp);
540
```

```
541         JMenuItem mntmAbout = new JMenuItem("About...");
542         mntmAbout.setAction(aboutAction);
543         mnHelp.add(mntmAbout);
544 
545         // ----------------------------------------------------------------------
546         // Ajout des contrôleurs aux widgets
547         // pour connaître les Listeners applicable à un widget
548         // dans WindowBuilder, sélectionnez un widger de l'UI puis Menu
549         // Contextuel -> Add event handler
550         // ----------------------------------------------------------------------
551         moveListener = new MoveShapeListener(drawingModel, history, infoLabel);
552         scaleListener = null; // TODO new ScaleShapeListener(drawingModel, history, infoLabel);
553         rotateListener = null; // TODO new RotateShapeListener(drawingModel, history, infoLabel);
554         selectionListener = new SelectionFigureListener(drawingModel, history, infoLabel);
555 
556         figureTypeCombobox.addItemListener(new ShapeItemListener(FigureType
557             .fromInteger(figureTypeCombobox.getSelectedIndex())));
558     }
559 
560     /**
561      * Action pour quitter l'application
562      * @author davidroussel
563      */
564     private class QuitAction extends AbstractAction // implements QuitHandler
565     {
566         /**
567          * Constructeur de l'action pour quitter l'application
568          * Met en place le raccourci clavier, l'icône et la description
569          * de l'action
570          */
571         public QuitAction()
572         {
573             putValue(NAME, "Quit");
574             /*
575              * Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()
576              * = InputEvent.CTRL_MASK on win/linux
577              * = InputEvent.META_MASK on mac os
578              */
579             putValue(ACCELERATOR_KEY,
580                     KeyStroke.getKeyStroke(KeyEvent.VK_Q, Toolkit.getDefaultToolkit().getMenuShortc
    utKeyMask()));
581             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Quit"));
582             putValue(SMALL_ICON, IconFactory.getIcon("Quit_small"));
583             putValue(SHORT_DESCRIPTION, "Quits the application");
584         }
585 
586         /**
587          * Opérations réalisées par l'action : Quitte l'application
588          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
589          * ou d'un item de menu
590          */
591         @Override
592         public void actionPerformed(ActionEvent e)
593         {
594             doQuit();
595         }
596 
597         /**
598          * Action réalisée pour quitter dans un {@link Action}
599          */
600         private void doQuit()
601         {
602             /*
603              * Action à effectuer lorsque l'action "undo" est cliquée :
604              * sortir avec un System.exit() (pas très propre, mais fonctionne)
605              */
606             System.exit(0);
607         }
608     }
609 
610     /**
611      * Action réalisée pour effacer la dernière action dans le dessin
612      */
613     private class UndoAction extends AbstractAction
614     {
615         /**
616          * Constructeur de l'action effacer la dernière action sur le dessin
617          * Met en place le raccourci clavier, l'icône et la description
618          * de l'action
619          */
620         public UndoAction()
621         {
622             putValue(NAME, "Undo");
623             putValue(ACCELERATOR_KEY,
624                     KeyStroke.getKeyStroke(KeyEvent.VK_Z,
625                                            Toolkit.getDefaultToolkit()
626                                                   .getMenuShortcutKeyMask()));
627             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Undo"));
628             putValue(SMALL_ICON, IconFactory.getIcon("Undo_small"));
629             putValue(SHORT_DESCRIPTION, "Undo last drawing");
```



```
630         }
631 
632         /**
633          * Opérations réalisées par l'action : Mise en place du dernier
634          * Memento enregistré dans la pile des undo
635          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
636          * ou d'un item de menu
637          */
638         @Override
639         public void actionPerformed(ActionEvent e)
640         {
641             // TODO Compléter ...
642         }
643     }
644 
645     /**
646      * Action réalisée pour refaire la dernière action (qui a été annulée)
647      * dans le dessin
648      */
649     private class RedoAction extends AbstractAction
650     {
651         public RedoAction()
652         {
653             putValue(NAME, "Redo");
654             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Redo"));
655             putValue(SMALL_ICON, IconFactory.getIcon("Redo_small"));
656             putValue(ACCELERATOR_KEY,
657                     KeyStroke.getKeyStroke(KeyEvent.VK_Z,
658                                            InputEvent.SHIFT_MASK
659                                                | Toolkit.getDefaultToolkit()
660                                                    .getMenuShortcutKeyMask()));
661             putValue(SHORT_DESCRIPTION, "Redo last drawing");
662         }
663 
664         /**
665          * Opérations réalisées par l'action : Mise en place du dernier
666          * Memento enregistré dans la pile des redo
667          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
668          * ou d'un item de menu
669          */
670         @Override
671         public void actionPerformed(ActionEvent e)
672         {
673             // TODO Compléter ...
674         }
675     }
676 
677     /**
678      * Action réalisée pour effacer toutes les figures du dessin
679      */
680     private class ClearAction extends AbstractAction
681     {
682         /**
683          * Constructeur de l'action pour effacer toutes les figures du dessin
684          * Met en place le raccourci clavier, l'icône et la description
685          * de l'action
686          */
687         public ClearAction()
688         {
689             putValue(NAME, "Clear");
690             putValue(ACCELERATOR_KEY,
691                     KeyStroke.getKeyStroke(KeyEvent.VK_X,
692                                            Toolkit.getDefaultToolkit()
693                                                   .getMenuShortcutKeyMask()));
694             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Clear"));
695             putValue(SMALL_ICON, IconFactory.getIcon("Clear_small"));
696             putValue(SHORT_DESCRIPTION, "Erase all drawings");
697         }
698 
699         /**
700          * Opérations réalisées par l'action : Effacement de toutes les figures
701          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
702          * ou d'un item de menu
703          */
704         @Override
705         public void actionPerformed(ActionEvent e)
706         {
707             /*
708              * Action à effectuer lorsque l'action "clear" est cliquée :
709              * Effacer toutes les figures du dessin
710              */
711             // TODO Compléter ...
712         }
713     }
714 
715     /**
716      * Action réalisée pour afficher la boite de dialogue "A propos ..."
717      */
718     private class AboutAction extends AbstractAction // implements AboutHandler
719     {
```

```
720         /**
721          * Constructeur de l'action pour afficher la boite de dialogue
722          * "A propos ..." Met en place le raccourci clavier, l'icône et la
723          * description de l'action
724          */
725         public AboutAction()
726         {
727             putValue(ACCELERATOR_KEY,
728                     KeyStroke.getKeyStroke(KeyEvent.VK_I,
729                                            Toolkit.getDefaultToolkit()
730                                                   .getMenuShortcutKeyMask()));
731             putValue(LARGE_ICON_KEY, IconFactory.getIcon("About"));
732             putValue(SMALL_ICON, IconFactory.getIcon("About_small"));
733             putValue(NAME, "About");
734             putValue(SHORT_DESCRIPTION, "App information");
735         }
736 
737         /**
738          * Opérations réalisées par l'action : Affichage d'une boite de dialogue
739          * Affichant des infos sur l'application
740          * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
741          * ou d'un item de menu
742          */
743         @Override
744         public void actionPerformed(ActionEvent e)
745         {
746             doAbout(e);
747         }
748 
749         /**
750          * Action réalisée pour "A propos" dans un {@link Action}
751          * @param e l'évènement ayant déclenché l'action
752          */
753         private void doAbout(EventObject e)
754         {
755             /*
756              * Action à effectuer lorsque l'action "about" est cliquée :
757              * Ouvrir un MessageDialog (JOptionPane.showMessageDialog(...)) de
758              * type JOptionPane.INFORMATION_MESSAGE
759              */
760             Object source = e.getSource();
761             Component component =
762                 source instanceof Component ? (Component) source : null;
763             JOptionPane.showMessageDialog(component,
764                                           EditorName,
765                                           "About ...",
766                                           JOptionPane.INFORMATION_MESSAGE);
767         }
768     }
769 
770     /**
771      * Action réalisée lorsque l'on passe en mode édition des figures
772      */
773     private class ToggleCreateEditAction extends AbstractAction
774     {
775         /**
776          * Liste des "boutons" pouvant déclencher cette action,
777          * De manière à ce que lorqu'un bouton déclenche l'action
778          * les autres boutons soient eux aussi mis dans l'état correspondant
779          * à l'action
780          */
781         private List<AbstractButton> buttons;
782 
783         /**
784          * Constructeur de l'action pour mettre en place ou enlever un filtre
785          * pour filtrer les types de figures
786          */
787         public ToggleCreateEditAction()
788         {
789             putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_TAB, InputEvent.ALT_MASK));
790             putValue(NAME, "Edition");
791             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Edition"));
792             putValue(SMALL_ICON, IconFactory.getIcon("Edition_small"));
793             putValue(SHORT_DESCRIPTION, "Édition des figures");
794 
795             buttons = new ArrayList<AbstractButton>();
796         }
797 
798         /**
799          * Ajout d'un bouton déclenchant cette action
800          * @param button le bouton à ajouter à la liste des boutons
801          * @return true si le bouton a été ajouté à la liste des boutons
802          * déclenchant cette action, false si le bouton était déjà présent
803          * dans la liste des actions et n'a pas été ajouté
804          */
805         public boolean registerButton(AbstractButton button)
806         {
807             if (¬buttons.contains(button))
808             {
809                 return buttons.add(button);
```



```
810             }
811             return false;
812         }
813 
814         /**
815          * Opérations réalisées par l'action : Changement de mode Création /
816          * Edition des figures
817          * @param event l'évènement déclenchant l'action. Peut provenir d'un
818          * bouton ou d'un item de menu
819          */
820         @Override
821         public void actionPerformed(ActionEvent event)
822         {
823             AbstractButton button = (AbstractButton) event.getSource();
824             boolean selected = button.getModel().isSelected();
825 
826             /*
827              * TODO Parcourir tous les "buttons" pour s'assurer qu'ils sont
828              * bien dans l'état voulu
829              */
830 
831             /*
832              * TODO
833              * Si on est en mode :
834              *  - Creation : on met en place le creationListener courant dans
835              *  drawingPanelpour créer la prochaine figure et on enlève de
836              *  drawingPanel tous les listeners pour modifier les figures
837              *  - Edition On retire le creationListener de drawingPanel puis on
838              *  met en places les listeners dans drawingPanel pour
839              *      - pouvoir sélectionner/désélectionner des figures
840              *      - déplacer des figures
841              *      - tourner des figures
842              *      - changer l'échelle des figures
843              *
844              */
845         }
846     }
847 
848 
849     /**
850      * Action réalisée pour filtrer ou pas le flux de figures
851      */
852     private class FilterAction extends AbstractAction
853     {
854         /**
855          * Constructeur de l'action pour mettre en place ou enlever un filtre
856          * pour filtrer les types de figures
857          */
858         public FilterAction()
859         {
860             putValue(NAME, "Filter");
861             putValue(LARGE_ICON_KEY, IconFactory.getIcon("Filter"));
862             putValue(SMALL_ICON, IconFactory.getIcon("Filter_small"));
863             putValue(SHORT_DESCRIPTION, "Set/unset filtering");
864             putValue(ACCELERATOR_KEY,
865                     KeyStroke.getKeyStroke(KeyEvent.VK_F,
866                                            Toolkit.getDefaultToolkit()
867                                                   .getMenuShortcutKeyMask()));
868 
869         }
870 
871         /**
872          * Opérations réalisées par l'action : Mise en place ou arrêt du
873          * filtrage des figures
874          * @param event l'évènement déclenchant l'action. Peut provenir d'un
875          * bouton ou d'un item de menu
876          */
877         @Override
878         public void actionPerformed(ActionEvent event)
879         {
880             AbstractButton button = (AbstractButton) event.getSource();
881             boolean selected = button.getModel().isSelected();
882 
883             // TODO Compléter ...
884         }
885     }
886 
887 
888     /**
889      * Action réalisée pour ajouter ou retirer un filtre de type de figure
890      */
891     private class ShapeFilterAction extends AbstractAction
892     {
893         /**
894          * Le type de figure
895          */
896         private FigureType type;
897 
898         /**
899          * Constructeur de l'action pour mettre en place ou enlever un filtre
```

```
900             * pour filtrer les types de figures
901             */
902            public ShapeFilterAction(FigureType type)
903            {
904                this.type = type;
905                String name = type.toString();
906                putValue(LARGE_ICON_KEY, IconFactory.getIcon(name));
907                putValue(SMALL_ICON, IconFactory.getIcon(name + "_small"));
908                putValue(NAME, name);
909                putValue(SHORT_DESCRIPTION, "Set/unset " + name + " filter");
910            }
911
912            /**
913             * Opérations réalisées par l'action : Ajout ou retrait d'un filtre
914             * concernant un type particulier de figure ({@link #type})
915             * @param event l'évènement déclenchant l'action. Peut provenir d'un
916             * bouton ou d'un item de menu
917             */
918            @Override
919            public void actionPerformed(ActionEvent event)
920            {
921                AbstractButton button = (AbstractButton) event.getSource();
922                boolean selected = button.getModel().isSelected();
923
924                // TODO Compléter ...
925            }
926        }
927
928        /**
929         * Action réalisée pour ajouter ou retirer un filtre de type trait de figure
930         */
931        private class LineFilterAction extends AbstractAction
932        {
933            /**
934             * Le type de trait de la figure
935             */
936            private LineType type;
937
938            /**
939             * Constructeur de l'action pour mettre en place ou enlever un filtre
940             * pour filtrer les types de figures
941             */
942            public LineFilterAction(LineType type)
943            {
944                this.type = type;
945                String name = type.toString();
946                putValue(LARGE_ICON_KEY, IconFactory.getIcon(name));
947                putValue(SMALL_ICON, IconFactory.getIcon(name + "_small"));
948                putValue(NAME, name);
949                putValue(SHORT_DESCRIPTION, "Set/unset " + name + " filter");
950            }
951
952            /**
953             * Opérations réalisées par l'action : Ajout ou retrait d'un filtre
954             * concernant le type de trait des figures
955             * @param event l'évènement déclenchant l'action. Peut provenir d'un
956             * bouton ou d'un item de menu
957             */
958            @Override
959            public void actionPerformed(ActionEvent event)
960            {
961                AbstractButton button = (AbstractButton) event.getSource();
962                boolean selected = button.getModel().isSelected();
963
964                // TODO Compléter ...
965            }
966        }
967
968        /**
969         * Action pour mettre en place un filtre basé sur la couleur de remplissage
970         * courante
971         */
972        private class FillColorFilterAction extends AbstractAction
973        {
974            /**
975             * Constructeur de l'action
976             * Met en place le raccourci clavier, l'icône et la description
977             * de l'action
978             */
979            public FillColorFilterAction()
980            {
981                putValue(NAME, "Fill Color");
982                putValue(LARGE_ICON_KEY, IconFactory.getIcon("FillColor"));
983                putValue(SMALL_ICON, IconFactory.getIcon("FillColor_small"));
984                putValue(SHORT_DESCRIPTION, "Set/Unset Fill Color Filter");
985            }
986
987            /**
988             * Opérations réalisées par l'action : Ajout ou retrait du filtre
989             * de couleur de remplissage en fonction de la couleur de remplissage
```



```
990             * courante.
991             * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
992             * ou d'un item de menu
993             */
994            @Override
995            public void actionPerformed(ActionEvent e)
996            {
997                AbstractButton button = (AbstractButton) e.getSource();
998                boolean selected = button.getModel().isSelected();
999
1000               // TODO Compléter ...
1001           }
1002       }
1003
1004       /**
1005        * Action pour mettre en place un filtre basé sur la couleur de trait
1006        * courante
1007        */
1008       private class EdgeColorFilterAction extends AbstractAction
1009       {
1010           /**
1011            * Constructeur de l'action
1012            * Met en place le raccourci clavier, l'icône et la description
1013            * de l'action
1014            */
1015           public EdgeColorFilterAction()
1016           {
1017               putValue(NAME, "Edge Color");
1018               putValue(LARGE_ICON_KEY, IconFactory.getIcon("EdgeColor"));
1019               putValue(SMALL_ICON, IconFactory.getIcon("EdgeColor_small"));
1020               putValue(SHORT_DESCRIPTION, "Set/Unset edge color filter");
1021           }
1022
1023           /**
1024            * Opérations réalisées par l'action : Ajout ou retrait d'un filtre
1025            * concernant la couleur de trait d'après la couleur de trait courante.
1026            * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
1027            * ou d'un item de menu
1028            */
1029           @Override
1030           public void actionPerformed(ActionEvent e)
1031           {
1032               AbstractButton button = (AbstractButton) e.getSource();
1033               boolean selected = button.getModel().isSelected();
1034
1035               // TODO Compléter ...
1036           }
1037       }
1038
1039       /**
1040        * Action réalisée pour détruire les figures sélectionnées
1041        * @author davidroussel
1042        */
1043       private class DeleteAction extends AbstractAction
1044       {
1045           public DeleteAction()
1046           {
1047               putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_X, 0));
1048               putValue(NAME, "Delete");
1049               putValue(LARGE_ICON_KEY, IconFactory.getIcon("Delete"));
1050               putValue(SMALL_ICON, IconFactory.getIcon("Delete_small"));
1051               putValue(SHORT_DESCRIPTION, "Delete selected figures");
1052           }
1053
1054           /**
1055            * Opérations réalisées par l'action : Retrait des figures sélectionnées.
1056            * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
1057            * ou d'un item de menu
1058            */
1059           @Override
1060           public void actionPerformed(ActionEvent e)
1061           {
1062               // TODO Compléter ...
1063           }
1064       }
1065
1066       /**
1067        * Action réalisée pour remonter les figures sélectionnées dans la liste
1068        * des figures
1069        */
1070       private class MoveUpAction extends AbstractAction
1071       {
1072           public MoveUpAction()
1073           {
1074               putValue(ACCELERATOR_KEY,
1075                       KeyStroke.getKeyStroke(KeyEvent.VK_UP,
1076                                              Toolkit.getDefaultToolkit()
1077                                                  .getMenuShortcutKeyMask()));
1078               putValue(NAME, "Up");
1079               putValue(LARGE_ICON_KEY, IconFactory.getIcon("MoveUp"));
```

```
1080                putValue(SMALL_ICON, IconFactory.getIcon("MoveUp_small"));
1081                putValue(SHORT_DESCRIPTION, "Move selected figures up");
1082            }
1083
1084            /**
1085             * Opérations réalisées par l'action : Déplacement des figures
1086             * sélectionnées en haut de la liste des figures.
1087             * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
1088             * ou d'un item de menu
1089             */
1090            @Override
1091            public void actionPerformed(ActionEvent e)
1092            {
1093                // TODO Compléter ...
1094            }
1095        }
1096
1097        /**
1098         * Action réalisée pour descendre les figures sélectionnées dans la liste
1099         * des figures
1100         */
1101        private class MoveDownAction extends AbstractAction
1102        {
1103            public MoveDownAction()
1104            {
1105                putValue(ACCELERATOR_KEY,
1106                        KeyStroke.getKeyStroke(KeyEvent.VK_DOWN,
1107                                               Toolkit.getDefaultToolkit()
1108                                                   .getMenuShortcutKeyMask()));
1109                putValue(NAME, "Down");
1110                putValue(LARGE_ICON_KEY, IconFactory.getIcon("MoveDown"));
1111                putValue(SMALL_ICON, IconFactory.getIcon("MoveDown_small"));
1112                putValue(SHORT_DESCRIPTION, "Move selected figures down");
1113            }
1114
1115            /**
1116             * Opérations réalisées par l'action : Déplacement des figures
1117             * sélectionnées en bas de la liste des figures.
1118             * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
1119             * ou d'un item de menu
1120             */
1121            @Override
1122            public void actionPerformed(ActionEvent e)
1123            {
1124                // TODO Compléter ...
1125            }
1126        }
1127
1128        /**
1129         * Action réalisée pour appliquer le style courant aux figures
1130         * sélectionnées,
1131         * A savoir :
1132         * <ul>
1133         * <li>La couleur de remplissage courante</li>
1134         * <li>La couleur de trait courante</li>
1135         * <li>Le type de trait courant (style et épaisseur)</li>
1136         * </ul>
1137         */
1138        private class StyleAction extends AbstractAction
1139        {
1140            public StyleAction()
1141            {
1142                putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke(KeyEvent.VK_S, 0));
1143                putValue(NAME, "Style");
1144                putValue(LARGE_ICON_KEY, IconFactory.getIcon("Style"));
1145                putValue(SMALL_ICON, IconFactory.getIcon("Style_small"));
1146                putValue(SHORT_DESCRIPTION,
1147                        "Apply current style to selected figures");
1148            }
1149
1150            /**
1151             * Opérations réalisées par l'action : Application du style courant (
1152             * couleur de remplissage, couleur de trait, type de trait et épaisseur
1153             * du trait) aux figures sélectionnées.
1154             * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
1155             * ou d'un item de menu
1156             */
1157            @Override
1158            public void actionPerformed(ActionEvent e)
1159            {
1160                // TODO Compléter ...
1161            }
1162        }
1163
1164        /**
1165         * Contrôleur d'évènement permettant de modifier le type de figures à
1166         * dessiner.
1167         * @note dépends de #drawingModel et #infoLabel qui doivent être non
1168         * null avant instanciation
1169         */
```



```
1170        private class ShapeItemListener implements ItemListener
1171        {
1172            /**
1173             * Constructeur valué du contrôleur.
1174             * Initialise le type de dessin dans {@link EditorFrame#drawingModel}
1175             * et crée le {@link AbstractCreationListener} correspondant.
1176             * @param initialIndex l'index du type de forme sélectionné afin de
1177             * mettre en place le bon creationListener dans le
1178             * {@link EditorFrame#drawingPanel}.
1179             */
1180            public ShapeItemListener(FigureType type)
1181            {
1182                // Mise en place du type de figure
1183                drawingModel.setFigureType(type);
1184
1185                // Mise en place du type de creationListener
1186                creationListener = type.getCreationListener(drawingModel,
1187                                                            history,
1188                                                            infoLabel);
1189                drawingPanel.addFigureListener(creationListener);
1190            }
1191
1192            @Override
1193            public void itemStateChanged(ItemEvent e)
1194            {
1195                JComboBox<?> items = (JComboBox<?>) e.getSource();
1196                int index = items.getSelectedIndex();
1197                int stateChange = e.getStateChange();
1198                FigureType figureType = FigureType.fromInteger(index);
1199                switch (stateChange)
1200                {
1201                    case ItemEvent.SELECTED:
1202                    {
1203                        // Mise en place d'un nouveau type de figure
1204                        drawingModel.setFigureType(figureType);
1205                        AbstractCreationListener newCreationListener =
1206                            figureType.getCreationListener(drawingModel,
1207                                                           history,
1208                                                           infoLabel);
1209                        if (operationMode == OperationMode.CREATION)
1210                        {
1211                            // Mise en place d'un nouveau type de creationListener
1212                            // Après avoir retiré l'ancien dans le drawingPanel
1213                            drawingPanel.removeFigureListener(creationListener);
1214                            drawingPanel.addFigureListener(newCreationListener);
1215                        }
1216                        creationListener = newCreationListener;
1217                        break;
1218                    }
1219                }
1220            }
1221        }
1222
1223        /**
1224         * Contrôleur d'évènements permettant de modifier la couleur du trait.
1225         * @note utilise #drawingModel qui doit être non null avant instanciation
1226         * @note A associer comme listener au JJ[Labeled]ComboBox des couleurs de
1227         * remplissage ou de trait
1228         */
1229        private class ColorItemListener implements ItemListener
1230        {
1231            /**
1232             * Ce à quoi s'applique la couleur choisie.
1233             * Soit au rmplissage, soit au trait.
1234             */
1235            private PaintToType applyTo;
1236
1237            /**
1238             * La dernière couleur choisie (pour le {@link JColorChooser})
1239             */
1240            private Color lastColor;
1241
1242            /**
1243             * Le tableau des couleurs possibles
1244             */
1245            private Paint[] colors;
1246
1247            /**
1248             * L'index de la couleur spéciale à choisir avec un
1249             * {@link JColorChooser}
1250             */
1251            private final int customColorIndex;
1252
1253            /**
1254             * L'index de la dernière couleur sélectionnée dans le combobox.
1255             * Afin de pouvoir y revenir si jamais le {@link JColorChooser} est
1256             * annulé.
1257             */
1258            private int lastSelectedIndex;
1259
```

```
1260        /**
1261         * la couleur choisie
1262         */
1263        private Paint paint;
1264
1265        /**
1266         * Constructeur du contrôleur d'évènements d'un combobox permettant
1267         * de choisir la couleur de remplissage
1268         * @param colors le tableau des couleurs possibles
1269         * @param selectedIndex l'index de l'élément actuellement sélectionné
1270         * @param customColorIndex l'index de la couleur spéciale parmis les
1271         * colors à définir à l'aide d'un {@link JColorChooser}.
1272         * @param applyTo Ce à quoi s'applique la couleur (le remplissage ou
1273         * bien le trait)
1274         */
1275        public ColorItemListener(Paint[] colors,
1276                                 int selectedIndex,
1277                                 int customColorIndex,
1278                                 PaintToType applyTo)
1279        {
1280            this.colors = colors;
1281            lastSelectedIndex = selectedIndex;
1282            this.customColorIndex = customColorIndex;
1283            this.applyTo = applyTo;
1284            lastColor = (Color) colors[selectedIndex];
1285            paint = colors[selectedIndex];
1286
1287            applyTo.applyPaintTo(paint, drawingModel);
1288        }
1289
1290        /**
1291         * Actions à réaliser lorsque l'élément sélectionné du combox change
1292         * @param e l'évènement de changement d'item du combobox
1293         */
1294        @Override
1295        public void itemStateChanged(ItemEvent e)
1296        {
1297            JComboBox<?> combo = (JComboBox<?>) e.getSource();
1298            int index = combo.getSelectedIndex();
1299
1300            if ((index ≥ 0) ∧ (index < colors.length))
1301            {
1302                if (e.getStateChange() ≡ ItemEvent.SELECTED)
1303                {
1304                    // New color has been selected
1305                    if (index ≡ customColorIndex) // Custom color from chooser
1306                    {
1307                        Paint chosenColor = PaintFactory
1308                            .getPaint(combo,
1309                                      "Choose " + applyTo.toString() + " Color",
1310                                      lastColor);
1311                        if (chosenColor ≠ null)
1312                        {
1313                            paint = chosenColor;
1314                        }
1315                        else
1316                        {
1317                            // ColorChooser has been cancelled we should go
1318                            // back to last selected index
1319                            combo.setSelectedIndex(lastSelectedIndex);
1320
1321                            // paint does not change
1322                        }
1323                    }
1324                    else // regular color
1325                    {
1326                        paint = colors[index];
1327                    }
1328
1329                    lastColor = (Color) paint;
1330                    applyTo.applyPaintTo(paint, drawingModel);
1331                }
1332                else if (e.getStateChange() ≡ ItemEvent.DESELECTED)
1333                {
1334                    // Old color has been delesected
1335                    if ((index ≥ 0) ∧ (index < customColorIndex))
1336                    {
1337                        lastColor = (Color) edgePaints[index];
1338                        lastSelectedIndex = index;
1339                    }
1340                }
1341            }
1342            else
1343            {
1344                System.err.println("Unknown " + applyTo.toString()
1345                    + " color index : " + index);
1346            }
1347        }
1348    }
1349
```



```
1350    /**
1351     * Contrôleur d'évènements permettant de modifier le type de trait (normal,
1352     * pointillé, sans trait)
1353     * @note utilise #drawingModel qui doit être non null avant instanciation
1354     * @note A associer comme listener au J[Labeled]Combobox des types de traits
1355     */
1356    private class EdgeTypeListener implements ItemListener
1357    {
1358        /**
1359         * Le type de trait à mettre en place
1360         */
1361        private LineType edgeType;
1362
1363        public EdgeTypeListener(LineType type)
1364        {
1365            edgeType = type;
1366            drawingModel.setEdgeType(edgeType);
1367        }
1368
1369        @Override
1370        public void itemStateChanged(ItemEvent e)
1371        {
1372            JComboBox<?> items = (JComboBox<?>) e.getSource();
1373            int index = items.getSelectedIndex();
1374
1375            if (e.getStateChange() ≡ ItemEvent.SELECTED)
1376            {
1377                // actions à réaliser lorsque le type de trait change
1378                LineType type = LineType.fromInteger(index);
1379                drawingModel.setEdgeType(type);
1380            }
1381        }
1382    }
1383
1384    /**
1385     * Contrôleur d'évènement permettant de modifier la taille du trait
1386     * en fonction des valeurs d'un {@link JSpinner}
1387     * @note à associer comme listener au JSpinner de l'épaisseur de trait
1388     */
1389    private class EdgeWidthListener implements ChangeListener
1390    {
1391        /**
1392         * Constructeur du contrôleur d'évènements contrôlant l'épaisseur du
1393         * trait
1394         * @param initialValue la valeur initiale de la largeur du trait à
1395         * appliquer au dessin (EditorFrame#drawingModel)
1396         */
1397        public EdgeWidthListener(int initialValue)
1398        {
1399            drawingModel.setEdgeWidth(initialValue);
1400        }
1401
1402        /**
1403         * Actions à réaliser lorsque la valeur du spinner change
1404         * @param e l'évènement de changement de valeur du spinner
1405         */
1406        @Override
1407        public void stateChanged(ChangeEvent e)
1408        {
1409            JSpinner spinner = (JSpinner) e.getSource();
1410            SpinnerNumberModel spinnerModel =
1411                (SpinnerNumberModel) spinner.getModel();
1412
1413            drawingModel.setEdgeWidth(spinnerModel.getNumber().floatValue());
1414        }
1415    }
1416
1417    /**
1418     * Action pour
1419     * @author davidroussel
1420     */
1421    @SuppressWarnings("unused")
1422    private class EmptyAction extends AbstractAction
1423    {
1424        /**
1425         * Constructeur de l'action pour ...
1426         * Met en place le raccourci clavier, l'icône et la description
1427         * de l'action
1428         */
1429        public EmptyAction()
1430        {
1431            String name = "XXX";
1432            putValue(NAME, name);
1433            /*
1434             * Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()
1435             * = InputEvent.CTRL_MASK on win/linux
1436             * = InputEvent.META_MASK on mac os
1437             */
1438            putValue(ACCELERATOR_KEY,
1439                     KeyStroke.getKeyStroke(KeyEvent.VK_X,
```

```
                                        Toolkit.getDefaultToolkit()
                                                .getMenuShortcutKeyMask()));
            putValue(LARGE_ICON_KEY, IconFactory.getIcon(name));
            putValue(SMALL_ICON, IconFactory.getIcon(name + "_small"));
            putValue(SHORT_DESCRIPTION, "Description de l'action");
        }

        /**
         * Opérations réalisées par l'action
         * @param e l'évènement déclenchant l'action. Peut provenir d'un bouton
         * ou d'un item de menu
         */
        @Override
        public void actionPerformed(ActionEvent e)
        {
            AbstractButton button = (AbstractButton) e.getSource();
            boolean selected = button.getModel().isSelected();

            // drawingModel.awesomeMethod(...)
        }
    }
}
```



```
package widgets;

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

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

import figures.Drawing;
import figures.Figure;
import figures.listeners.AbstractFigureListener;
import figures.listeners.creation.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
     * pointeur de la souris
     */
    private JLabel coordLabel;

    /**
     * L'{@link InfoPanel} dans lequel afficher les informations à propos de
```

```
91       * la figure sous le curseur.
92       */
93      private InfoPanel infoPanel;
94
95      /**
96       * Chaîne de caractère à afficher par défaut dans le {@link #coordLabel}
97       */
98      public final static String defaultCoordString = new String("x: ___ y: ___");
99
100     /**
101      * Le formatteur à utiliser pour formater les nombres dans le
102      * {@link #coordLabel} et dans l'{@link #infoPanel}
103      */
104     private final static DecimalFormat coordFormat = new DecimalFormat("000");
105
106     /**
107      * état indiquant s'il faut envoyer les coordonnées de la souris ou la
108      * figure au dessus de laquelle se trouve la souris. Lorsque le curseur sort
109      * du widget (mouseExited) on cesse d'envoyer les coordonnées de la souris
110      * et lorsqu'elle entre (mouseEntered) on recommence à envoyer les
111      * coordonnées de la souris.
112      */
113     private boolean sendInfoState;
114
115     /**
116      * Constructeur de la zone de dessin à partir d'un modèle de dessin.
117      *
118      * @param drawing le modèle de dessin
119      * @param coordLabel le label à mettre à jour avec les coordonnées du
120      *            curseur de la souris
121      * @param infoPanel le panneau d'information des figures à mettre à jour
122      *            avec les informations relative à la figure située sous le
123      *            curseur de la souris
124      */
125     public DrawingPanel(Drawing drawing, JLabel coordLabel, InfoPanel infoPanel)
126     {
127         setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
128         size = new Dimension(800, 600);
129         setPreferredSize(size);
130         addComponentListener(new ResizeListener());
131
132         setBackground(Color.WHITE);
133         setLayout(null);
134         setDoubleBuffered(true);
135
136         drawingModel = drawing;
137         if (drawing ≠ null)
138         {
139             drawingModel.addObserver(this);
140         }
141         else
142         {
143             System.err.println("DrawingPanel caution: null drawing");
144         }
145
146         this.coordLabel = coordLabel;
147
148         if (this.coordLabel ≠ null)
149         {
150             this.coordLabel.setText(defaultCoordString);
151         }
152         else
153         {
154             System.err.println("DrawingPanel : null coordLabel");
155         }
156
157         this.infoPanel = infoPanel;
158
159         if (this.infoPanel ≠ null)
160         {
161             this.infoPanel.resetLabels();
162         }
163         else
164         {
165             System.err.println("DrawingPanel : null infoPanel");
166         }
167
168         // DrawingPanel est son propre listener d'évènements souris
169         addMouseListener(this);
170         addMouseMotionListener(this);
171     }
172
173     @Override
174     protected void finalize() throws Throwable
175     {
176         drawingModel.deleteObserver(this);
177         super.finalize();
178     }
179
180     /**
```



```
181      * Accès à la taille effective du panel qui peut changer si celui-ci est
182      * agrandi (avec la fenêtre dans lequel il est par exemple). Cette méthode
183      * permet d'ajuster les scrollbars d'un container qui contiendrait ce panel
184      * lorsque la taille de celui-ci change.
185      *
186      * @return la taille effective du panel de dessin
187      * @see javax.swing.JComponent#getPreferredSize()
188      */
189     @Override
190     public Dimension getPreferredSize()
191     {
192         return size;
193     }
194
195     /**
196      * Mise en place du modèle de dessin. Met en place un nouveau modèle et s'il
197      * est non null ajoute ce panel comme observateur du modèle
198      *
199      * @param drawing le modèle de dessin à mettre en place
200      */
201     public void setDrawing(Drawing drawing)
202     {
203         // retrait du précédent modèle de dessin (s'il existe)
204         if (drawingModel ≠ null)
205         {
206             drawing.deleteObserver(this);
207         }
208
209         // Mise en place du nouveau modèle de dessin
210         drawingModel = drawing;
211         if (drawingModel ≠ null)
212         {
213             drawingModel.addObserver(this);
214         }
215     }
216
217     /**
218      * Mise en place du label dans lequel afficher les coordonnées du pointeur
219      * de la souris.
220      *
221      * @param coordLabel le label dans lequel afficher les coordonnées du
222      *            pointeur de la souris.
223      */
224     public void setCoordLabel(JLabel coordLabel)
225     {
226         this.coordLabel = coordLabel;
227     }
228
229     /**
230      * Mise en place du panel d'information dans lequel afficher les infos sur
231      * la figure située sous le curseur
232      *
233      * @param infoPanel l'{@link InfoPanel} à mettre en place
234      */
235     public void setInfoPanel(InfoPanel infoPanel)
236     {
237         this.infoPanel = infoPanel;
238     }
239
240     /**
241      * Dessin du panel. Effacement ce celui-ci puis dessin des figures.
242      * @param g le contexte graphique
243      * @see javax.swing.JComponent#paintComponent(java.awt.Graphics)
244      */
245     @Override
246     protected void paintComponent(Graphics g)
247     {
248         super.paintComponent(g); // Inutile
249
250         // caractéristiques graphiques : mise en place de l'antialiasing
251         Graphics2D g2D = (Graphics2D) g;
252         g2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
253                              RenderingHints.VALUE_ANTIALIAS_ON);
254
255         // taille de la zone de dessin
256         Dimension d = getSize();
257         // on commence par effacer le fond
258         g2D.setColor(getBackground());
259         g2D.fillRect(0, 0, d.width, d.height);
260
261         // Puis on dessine l'ensemble des figures
262         if (drawingModel ≠ null)
263         {
264             /*
265              * Application d'un Consumer<Figure> en tant que lambda expression
266              * sur le flux (éventuellement filtré) des figures permettant
267              * de dessiner les figures
268              */
269             drawingModel.stream().forEach((Figure f) → f.draw(g2D));
270
```

```
271             /*
272              * Soulignement des figures sélectionnées (s'il y en a).
273              * Le soulignement est séparé du dessin des figures elles mêmes
274              * de manière à apparaître par dessus les figures dessinées
275              */
276             if (drawingModel.hasSelection())
277             {
278                 drawingModel.stream().forEach((Figure f) → f.drawSelection(g2D));
279             }
280         }
281         else
282         {
283             System.err.println(getClass().getSimpleName() + "::paintComponent : null model");
284         }
285     }
286
287     /**
288      * Mise en place d'un nouveau listener de figure
289      *
290      * @param fl le nouveau listener
291      */
292     public void addFigureListener(AbstractFigureListener fl)
293     {
294         if (fl ≠ null)
295         {
296             addMouseListener(fl);
297             addMouseMotionListener(fl);
298             // System.out.println("CreationListener " + cl + " added");
299         }
300         else
301         {
302             System.err.println("DrawingPanel.addFigureListener(null)");
303         }
304     }
305
306     /**
307      * Retrait d'un listener de figure
308      *
309      * @param fl le creationListener à retirer
310      */
311     public void removeFigureListener(AbstractFigureListener fl)
312     {
313         if (fl ≠ null)
314         {
315             removeMouseListener(fl);
316             removeMouseMotionListener(fl);
317             // System.out.println("CreationListener " + cl + " removed");
318         }
319     }
320
321     /**
322      * Mise à jour déclenchée par un {@link Observable#notifyObservers()} : en
323      * l'occurence le modèle de dessin ({@link Drawing}) lorsque celui ci est
324      * modifié. Cette mise à jour déclenche une requête de redessin du panel.
325      *
326      * @param observable l'observable ayant déclenché cette MAJ
327      * @param data les données (evt) transmises par l'observable [non utilisé ici]
328      * @see java.util.Observer#update(java.util.Observable, java.lang.Object)
329      */
330     @Override
331     public void update(Observable observable, Object data)
332     {
333         if (observable instanceof Drawing)
334         {
335             // Le modèle à changé il faut redessiner les figures
336             repaint();
337         }
338     }
339
340     /**
341      * Rafraichissement des panneaux d'information lors du déplacement de la
342      * souris
343      *
344      * @param e l'évènement souris associé
345      */
346     @Override
347     public void mouseDragged(MouseEvent e)
348     {
349         // Déplacement de la souris (btn enfoncé) : MAJ des coordonnées
350         // de la souris dans le coordLabel et infoPanel
351         refreshCoordLabel(e.getPoint());
352         refreshInfoPanel(e.getPoint());
353     }
354
355     /**
356      * Rafraichissement des panneaux d'information lors du déplacement (bouton
357      * enfoncé) de la souris
358      *
359      * @param e l'évènement souris associé
360      */
```



```
361     @Override
362     public void mouseMoved(MouseEvent e)
363     {
364         // Déplacement de la souris : MAJ des coordonnées
365         // de la souris dans le coordLabel et infoPanel
366         Point p = e.getPoint();
367         refreshCoordLabel(p);
368         refreshInfoPanel(p);
369     }
370
371     @Override
372     public void mouseClicked(MouseEvent e)
373     {
374         // Rien
375     }
376
377     /**
378      * Reprise du rafraichissement des panneaux d'information lorsque la souris
379      * rentre dans ce panel.
380      *
381      * @param e l'évènement souris associé
382      */
383     @Override
384     public void mouseEntered(MouseEvent e)
385     {
386         sendInfoState = true;
387         refreshCoordLabel(e.getPoint());
388         refreshInfoPanel(e.getPoint());
389     }
390
391     /**
392      * Arrêt du rafraichissement des panneaux d'information et effacement de ces
393      * panneaux lorsque la souris sort du panel.
394      *
395      * @param e l'évènement souris associé
396      */
397     @Override
398     public void mouseExited(MouseEvent e)
399     {
400         // Rien si ce n'est de remettre les coordonnés dans la barre d'état
401         // à x = ___ y = ___
402         sendInfoState = false;
403         refreshCoordLabel(e.getPoint());
404         infoPanel.resetLabels();
405     }
406
407     @Override
408     public void mousePressed(MouseEvent e)
409     {
410         // Rien
411     }
412
413     @Override
414     public void mouseReleased(MouseEvent e)
415     {
416         // Rien
417     }
418
419     /**
420      * Rafraichissement du {@link #coordLabel} (s'il est non null) avec de
421      * nouvelles coordonnées ou bien avec la {@link #defaultCoordString} si l'on
422      * affiche pas les coordonnées
423      *
424      * @param x l'abcisse des coordonnées à afficher
425      * @param y l'ordonnée des coordonnées à afficher
426      */
427     private void refreshCoordLabel(Point p)
428     {
429         if ((coordLabel ≠ null) ∧ (p ≠ null))
430         {
431             if (sendInfoState)
432             {
433                 String xs = coordFormat.format(p.getX());
434                 String ys = coordFormat.format(p.getY());
435                 coordLabel.setText("x:" + xs + "y:" + ys);
436             }
437             else
438             {
439                 coordLabel.setText(defaultCoordString);
440             }
441         }
442     }
443
444     /**
445      * Rafraichissement du panneau d'information {@link #infoPanel}
446      *
447      * @param p la position du curseur pour déclencher la recherche de figures
448      *            sous ce curseur
449      */
450     private void refreshInfoPanel(Point2D p)
```

```
451     {
452         if ((infoPanel ≠ null) ∧ sendInfoState)
453         {
454             Figure selectedFigure = drawingModel.getFigureAt(p);
455 
456             if (selectedFigure ≠ null)
457             {
458                 infoPanel.updateLabels(selectedFigure);
459             }
460             else
461             {
462                 infoPanel.resetLabels();
463             }
464         }
465     }
466 }
```



```
1  package widgets.enums;
2  
3  /**
4   * Différents modes de fonctionnement de l'UI
5   * @author davidroussel
6   */
7  public enum OperationMode
8  {
9      /**
10      * Creation mode dans le quel on crée de nouvelles figures
11      */
12     CREATION,
13 
14     /**
15      * Transformation mode dans lequel on effectue des transformations
16      * géométriques (déplacement, rotation, facteur d'échelle) sur
17      * les figures sélectionnées
18      */
19     TRANSFORMATION;
20 
21     /**
22      * Nombre d'éléments dans cet enum
23      */
24     public static final int NbOperationModes = 2;
25 
26     /**
27      * Conversion d'un entier en {@link OperationMode}
28      *
29      * @param i l'entier à convertir en {@link OperationMode}
30      * @return l'OperationMode correspondant à l'entier
31      */
32     public static OperationMode fromInteger(int i)
33     {
34         switch (i)
35         {
36             case 0:
37                 return CREATION;
38             case 1:
39                 return TRANSFORMATION;
40             default:
41                 return CREATION;
42         }
43     }
44 
45     /**
46      * Index du mode
47      * @return l'index du mode
48      * @throws AssertionError si le mode est inconnu
49      */
50     public int toInteger() throws AssertionError
51     {
52         switch (this)
53         {
54             case CREATION:
55                 return 0;
56             case TRANSFORMATION:
57                 return 1;
58         }
59 
60         throw new AssertionError("OperationMode Unknown assertion " + this);
61     }
62 
63     /**
64      * Représentation sous forme de chaine de caractères
65      * @return une chaine de caractères représentant la valeur de cet enum
66      * @throws AssertionError si le mode est inconnu
67      */
68     @Override
69     public String toString() throws AssertionError
70     {
71         switch (this)
72         {
73             case CREATION:
74                 return new String("Creation");
75             case TRANSFORMATION:
76                 return new String("Edition");
77         }
78 
79         throw new AssertionError("OperationMode Unknown assertion " + this);
80     }
81 
82 
83     /**
84      * Mode suivant dans l'ordre des modes
85      * @return le mode suivant le mode courant
86      * @throws AssertionError si le mode est inconnu
87      */
88     public OperationMode nextMode() throws AssertionError
89     {
90         switch (this)
```

```
91              {
92                  case CREATION:
93                      return TRANSFORMATION;
94                  case TRANSFORMATION:
95                      return CREATION;
96              }
97
98              throw new AssertionError("OperationMode Unknown assertion " + this);
99          }
100
101 }
102
```



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

```java
package widgets.enums;

/**
 * Les types d'arbre pour représenter les figures dans un {@link javax.swing.JTree}
 * @author davidroussel
 */
public enum TreeType
{
    /**
     * Simple liste de figures
     */
    FIGURE,
    /**
     * Groupement des figures par type de figure
     */
    FIGURE_TYPE,
    /**
     * Groupement des figures par type de couleur de remplissage
     */
    FILL_COLOR,
    /**
     * Groupement des figures par type de couleur de trait
     */
    EDGE_COLOR,
    /**
     * Groupement des figures par type de trait
     */
    EDGE_TYPE;


    /**
     * Nombre d'éléments dans cet enum
     */
    public static final int NbTreeTypes = 5;

    /**
     * Conversion d'un entier en {@link TreeType}
     *
     * @param i l'entier à convertir en TreeType
     * @return le TreeType correspondant à l'entier
     */
    public static TreeType fromInteger(int i)
    {
        switch (i)
        {
            case 0:
                return FIGURE;
            case 1:
                return FIGURE_TYPE;
            case 2:
                return FILL_COLOR;
            case 3:
                return EDGE_COLOR;
            case 4:
                return EDGE_TYPE;
            default:
                return FIGURE;
        }
    }

    /**
     * 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 FIGURE:
                return new String("Figure");
            case FIGURE_TYPE:
                return new String("Figure Type");
            case FILL_COLOR:
                return new String("Fill Color");
            case EDGE_COLOR:
                return new String("Edge Color");
            case EDGE_TYPE:
                return new String("Edge Type");
        }

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



```java
package widgets;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.Paint;
import java.awt.geom.Point2D;
import java.awt.geom.Rectangle2D;
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.SwingConstants;
import javax.swing.border.LineBorder;

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

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
     */
```

```
91      private JLabel lblTlx;
92
93      /**
94       * Le label contenant l'ordonnée du point en haut à gauche de la figure
95       */
96      private JLabel lblTly;
97
98      /**
99       * Le label contenant l'abcisse du point en bas à droite de la figure
100      */
101     private JLabel lblBrx;
102
103     /**
104      * Le label contenant l'ordonnée du point en bas à droite de la figure
105      */
106     private JLabel lblBry;
107
108     /**
109      * Le label contenant la largeur de la figure
110      */
111     private JLabel lblDx;
112
113     /**
114      * Le label contenant la hauteur de la figure
115      */
116     private JLabel lblDy;
117
118     /**
119      * Le label contenant l'abcisse du barycentre de la figure
120      */
121     private JLabel lblCx;
122
123     /**
124      * Le label contenant l'ordonnée du barycentre de la figure
125      */
126     private JLabel lblCy;
127
128     /**
129      * Create the panel.
130      */
131     public InfoPanel()
132     {
133         // ----------------------------------------------------------------------
134         // Initialisation des maps
135         // ----------------------------------------------------------------------
136         figureIcons = new HashMap<FigureType, ImageIcon>();
137         for (int i = 0; i < FigureType.NbFigureTypes; i++)
138         {
139             FigureType type = FigureType.fromInteger(i);
140             figureIcons.put(type, IconFactory.getIcon(type.toString()));
141         }
142
143         paintIcons = new HashMap<Paint, ImageIcon>();
144         String[] colorStrings = {
145             "Black",
146             "Blue",
147             "Cyan",
148             "Green",
149             "Magenta",
150             "None",
151             "Orange",
152             "Others",
153             "Red",
154             "White",
155             "Yellow"
156         };
157
158         for (int i = 0; i < colorStrings.length; i++)
159         {
160             Paint paint = PaintFactory.getPaint(colorStrings[i]);
161             if (paint ≠ null)
162             {
163                 paintIcons.put(paint, IconFactory.getIcon(colorStrings[i]));
164             }
165         }
166
167         lineTypeIcons = new HashMap<LineType, ImageIcon>();
168         for (int i = 0; i < LineType.NbLineTypes; i++)
169         {
170             LineType type = LineType.fromInteger(i);
171             lineTypeIcons.put(type,  IconFactory.getIcon(type.toString()));
172         }
173
174         // ----------------------------------------------------------------------
175         // Création de l'UI
176         // ----------------------------------------------------------------------
177         setBorder(new LineBorder(new Color(0, 0, 0), 1, true));
178         GridBagLayout gridBagLayout = new GridBagLayout();
179         gridBagLayout.columnWidths = new int[] {80, 60, 60};
180         gridBagLayout.rowHeights = new int[] {30, 32, 32, 32, 20, 20, 20, 20, 20};
```



```
181         gridBagLayout.columnWeights = new double[]{0.0, 0.0, 0.0};
182         gridBagLayout.rowWeights = new double[]{0.0, 0.0, 0.0, 0.0};
183         setLayout(gridBagLayout);
184
185         lblFigureName = new JLabel("Figure Name");
186         lblFigureName.setHorizontalAlignment(SwingConstants.CENTER);
187         GridBagConstraints gbc_lblFigureName = new GridBagConstraints();
188         gbc_lblFigureName.insets = new Insets(5, 5, 5, 0);
189         gbc_lblFigureName.gridwidth = 3;
190         gbc_lblFigureName.gridx = 0;
191         gbc_lblFigureName.gridy = 0;
192         add(lblFigureName, gbc_lblFigureName);
193
194         JLabel lblType = new JLabel("type");
195         GridBagConstraints gbc_lblType = new GridBagConstraints();
196         gbc_lblType.anchor = GridBagConstraints.EAST;
197         gbc_lblType.insets = new Insets(0, 0, 5, 5);
198         gbc_lblType.gridx = 0;
199         gbc_lblType.gridy = 1;
200         add(lblType, gbc_lblType);
201
202         lblTypeicon = new JLabel(IconFactory.getIcon("Polygon"));
203         lblTypeicon.setHorizontalAlignment(SwingConstants.CENTER);
204         GridBagConstraints gbc_lblTypeicon = new GridBagConstraints();
205         gbc_lblTypeicon.insets = new Insets(0, 0, 5, 0);
206         gbc_lblTypeicon.gridwidth = 2;
207         gbc_lblTypeicon.gridx = 1;
208         gbc_lblTypeicon.gridy = 1;
209         add(lblTypeicon, gbc_lblTypeicon);
210
211         JLabel lblFill = new JLabel("fill");
212         GridBagConstraints gbc_lblFill = new GridBagConstraints();
213         gbc_lblFill.anchor = GridBagConstraints.EAST;
214         gbc_lblFill.insets = new Insets(0, 0, 5, 5);
215         gbc_lblFill.gridx = 0;
216         gbc_lblFill.gridy = 2;
217         add(lblFill, gbc_lblFill);
218
219         lblFillcolor = new JLabel(IconFactory.getIcon("White"));
220         GridBagConstraints gbc_lblFillcolor = new GridBagConstraints();
221         gbc_lblFillcolor.gridwidth = 2;
222         gbc_lblFillcolor.insets = new Insets(0, 0, 5, 0);
223         gbc_lblFillcolor.gridx = 1;
224         gbc_lblFillcolor.gridy = 2;
225         add(lblFillcolor, gbc_lblFillcolor);
226
227         JLabel lblStroke = new JLabel("stroke");
228         GridBagConstraints gbc_lblStroke = new GridBagConstraints();
229         gbc_lblStroke.anchor = GridBagConstraints.EAST;
230         gbc_lblStroke.insets = new Insets(0, 0, 5, 5);
231         gbc_lblStroke.gridx = 0;
232         gbc_lblStroke.gridy = 3;
233         add(lblStroke, gbc_lblStroke);
234
235         lblEdgecolor = new JLabel(IconFactory.getIcon("Black"));
236         GridBagConstraints gbc_lblStrokecolor = new GridBagConstraints();
237         gbc_lblStrokecolor.insets = new Insets(0, 0, 5, 5);
238         gbc_lblStrokecolor.gridx = 1;
239         gbc_lblStrokecolor.gridy = 3;
240         add(lblEdgecolor, gbc_lblStrokecolor);
241
242         lblStroketype = new JLabel(IconFactory.getIcon("Solid"));
243         GridBagConstraints gbc_lblStroketype = new GridBagConstraints();
244         gbc_lblStroketype.insets = new Insets(0, 0, 5, 0);
245         gbc_lblStroketype.gridx = 2;
246         gbc_lblStroketype.gridy = 3;
247         add(lblStroketype, gbc_lblStroketype);
248
249         JLabel lblX = new JLabel("x");
250         lblX.setFont(lblX.getFont().deriveFont(lblX.getFont().getSize() - 3f));
251         GridBagConstraints gbc_lblX = new GridBagConstraints();
252         gbc_lblX.insets = new Insets(0, 0, 5, 5);
253         gbc_lblX.gridx = 1;
254         gbc_lblX.gridy = 4;
255         add(lblX, gbc_lblX);
256
257         JLabel lblY = new JLabel("y");
258         lblY.setFont(lblY.getFont().deriveFont(lblY.getFont().getSize() - 3f));
259         GridBagConstraints gbc_lblY = new GridBagConstraints();
260         gbc_lblY.insets = new Insets(0, 0, 5, 0);
261         gbc_lblY.gridx = 2;
262         gbc_lblY.gridy = 4;
263         add(lblY, gbc_lblY);
264
265         JLabel lblTopLeft = new JLabel("top left");
266         lblTopLeft.setFont(lblTopLeft.getFont().deriveFont(lblTopLeft.getFont().getSize() - 3f));
267         GridBagConstraints gbc_lblTopLeft = new GridBagConstraints();
268         gbc_lblTopLeft.anchor = GridBagConstraints.EAST;
269         gbc_lblTopLeft.insets = new Insets(0, 0, 5, 5);
270         gbc_lblTopLeft.gridx = 0;
```

```
271         gbc_lblTopLeft.gridy = 5;
272         add(lblTopLeft, gbc_lblTopLeft);
273 
274         lblTlx = new JLabel("tlx");
275         lblTlx.setFont(lblTlx.getFont().deriveFont(lblTlx.getFont().getSize() - 3f));
276         GridBagConstraints gbc_lblTlx = new GridBagConstraints();
277         gbc_lblTlx.insets = new Insets(0, 0, 5, 5);
278         gbc_lblTlx.gridx = 1;
279         gbc_lblTlx.gridy = 5;
280         add(lblTlx, gbc_lblTlx);
281 
282         lblTly = new JLabel("tly");
283         lblTly.setFont(lblTly.getFont().deriveFont(lblTly.getFont().getSize() - 3f));
284         GridBagConstraints gbc_lblTly = new GridBagConstraints();
285         gbc_lblTly.insets = new Insets(0, 0, 5, 0);
286         gbc_lblTly.gridx = 2;
287         gbc_lblTly.gridy = 5;
288         add(lblTly, gbc_lblTly);
289 
290         JLabel lblBottomRight = new JLabel("bottom right");
291         lblBottomRight.setFont(lblBottomRight.getFont().deriveFont(lblBottomRight.getFont().getSize(
    ) - 3f));
292         GridBagConstraints gbc_lblBottomRight = new GridBagConstraints();
293         gbc_lblBottomRight.anchor = GridBagConstraints.EAST;
294         gbc_lblBottomRight.insets = new Insets(0, 0, 5, 5);
295         gbc_lblBottomRight.gridx = 0;
296         gbc_lblBottomRight.gridy = 6;
297         add(lblBottomRight, gbc_lblBottomRight);
298 
299         lblBrx = new JLabel("brx");
300         lblBrx.setFont(lblBrx.getFont().deriveFont(lblBrx.getFont().getSize() - 3f));
301         GridBagConstraints gbc_lblBrx = new GridBagConstraints();
302         gbc_lblBrx.insets = new Insets(0, 0, 5, 5);
303         gbc_lblBrx.gridx = 1;
304         gbc_lblBrx.gridy = 6;
305         add(lblBrx, gbc_lblBrx);
306 
307         lblBry = new JLabel("bry");
308         lblBry.setFont(lblBry.getFont().deriveFont(lblBry.getFont().getSize() - 3f));
309         GridBagConstraints gbc_lblBry = new GridBagConstraints();
310         gbc_lblBry.insets = new Insets(0, 0, 5, 0);
311         gbc_lblBry.gridx = 2;
312         gbc_lblBry.gridy = 6;
313         add(lblBry, gbc_lblBry);
314 
315         JLabel lblDimensions = new JLabel("dimensions");
316         lblDimensions.setFont(lblDimensions.getFont().deriveFont(lblDimensions.getFont().getSize() -
    3f));
317         GridBagConstraints gbc_lblDimensions = new GridBagConstraints();
318         gbc_lblDimensions.anchor = GridBagConstraints.EAST;
319         gbc_lblDimensions.insets = new Insets(0, 0, 5, 5);
320         gbc_lblDimensions.gridx = 0;
321         gbc_lblDimensions.gridy = 7;
322         add(lblDimensions, gbc_lblDimensions);
323 
324         lblDx = new JLabel("dx");
325         lblDx.setFont(lblDx.getFont().deriveFont(lblDx.getFont().getSize() - 3f));
326         GridBagConstraints gbc_lblDx = new GridBagConstraints();
327         gbc_lblDx.insets = new Insets(0, 0, 5, 5);
328         gbc_lblDx.gridx = 1;
329         gbc_lblDx.gridy = 7;
330         add(lblDx, gbc_lblDx);
331 
332         lblDy = new JLabel("dy");
333         lblDy.setFont(lblDy.getFont().deriveFont(lblDy.getFont().getSize() - 3f));
334         GridBagConstraints gbc_lblDy = new GridBagConstraints();
335         gbc_lblDy.insets = new Insets(0, 0, 5, 0);
336         gbc_lblDy.gridx = 2;
337         gbc_lblDy.gridy = 7;
338         add(lblDy, gbc_lblDy);
339 
340         JLabel lblCenter = new JLabel("center");
341         lblCenter.setFont(lblCenter.getFont().deriveFont(lblCenter.getFont().getSize() - 3f));
342         GridBagConstraints gbc_lblCenter = new GridBagConstraints();
343         gbc_lblCenter.anchor = GridBagConstraints.EAST;
344         gbc_lblCenter.insets = new Insets(0, 0, 0, 5);
345         gbc_lblCenter.gridx = 0;
346         gbc_lblCenter.gridy = 8;
347         add(lblCenter, gbc_lblCenter);
348 
349         lblCx = new JLabel("cx");
350         lblCx.setFont(lblCx.getFont().deriveFont(lblCx.getFont().getSize() - 3f));
351         GridBagConstraints gbc_lblCx = new GridBagConstraints();
352         gbc_lblCx.insets = new Insets(0, 0, 0, 5);
353         gbc_lblCx.gridx = 1;
354         gbc_lblCx.gridy = 8;
355         add(lblCx, gbc_lblCx);
356 
357         lblCy = new JLabel("cy");
358         lblCy.setFont(lblCy.getFont().deriveFont(lblCy.getFont().getSize() - 3f));
```



```
359         GridBagConstraints gbc_lblCy = new GridBagConstraints();
360         gbc_lblCy.gridx = 2;
361         gbc_lblCy.gridy = 8;
362         add(lblCy, gbc_lblCy);
363 
364     }
365 
366     /**
367      * Mise à jour de tous les labels avec les informations de figure
368      * @param figure la figure dont il faut extraire les informations
369      */
370     public void updateLabels(Figure figure)
371     {
372         // titre de la figure
373         lblFigureName.setText(figure.toString());
374 
375         // Icône du type de figure
376         lblTypeicon.setIcon(figureIcons.get(figure.getType()));
377 
378         // Icône de la couleur de remplissage
379         ImageIcon fillColorIcon = paintIcons.get(figure.getFillPaint());
380         if (fillColorIcon == null)
381         {
382             fillColorIcon = IconFactory.getIcon("Others");
383         }
384         lblFillcolor.setIcon(fillColorIcon);
385 
386         // Icône de la couleur de trait
387         ImageIcon edgeColorIcon = paintIcons.get(figure.getEdgePaint());
388         if (edgeColorIcon == null)
389         {
390             edgeColorIcon = IconFactory.getIcon("Others");
391         }
392         lblEdgecolor.setIcon(edgeColorIcon);
393 
394         // Icône du type de trait
395         BasicStroke stroke = figure.getStroke();
396         ImageIcon lineTypeIcon = null;
397         if (stroke == null)
398         {
399             lineTypeIcon = lineTypeIcons.get(LineType.NONE);
400         }
401         else
402         {
403             float[] dashArray = stroke.getDashArray();
404             if (dashArray == null)
405             {
406                 lineTypeIcon = lineTypeIcons.get(LineType.SOLID);
407             }
408             else
409             {
410                 lineTypeIcon = lineTypeIcons.get(LineType.DASHED);
411             }
412         }
413         lblStroketype.setIcon(lineTypeIcon);
414 
415         // Données numériques
416         Rectangle2D bounds = figure.getBounds2D();
417         Point2D center = figure.getCenter();
418 
419         double minX = bounds.getMinX();
420         double maxX = bounds.getMaxX();
421         double minY = bounds.getMinY();
422         double maxY = bounds.getMaxY();
423         double width = maxX - minX;
424         double height = maxY - minY;
425 
426         lblTlx.setText(coordFormat.format(minX));
427         lblTly.setText(coordFormat.format(minY));
428         lblBrx.setText(coordFormat.format(maxX));
429         lblBry.setText(coordFormat.format(maxY));
430 
431         lblDx.setText(coordFormat.format(width));
432         lblDy.setText(coordFormat.format(height));
433 
434         lblCx.setText(coordFormat.format(center.getX()));
435         lblCy.setText(coordFormat.format(center.getY()));
436     }
437 
438     /**
439      * Effacement de tous les labels
440      */
441     public void resetLabels()
442     {
443         // titre de la figure
444         lblFigureName.setText(emptyString);
445 
446         // Icône du type de figure
447         lblTypeicon.setIcon(emptyIcon);
448 
```

```
449         // Icône de la couleur de remplissage
450         lblFillcolor.setIcon(emptyIcon);
451 
452         // Icône de la couleur de trait
453         lblEdgecolor.setIcon(emptyIcon);
454 
455         // Icône du type de trait
456         lblStroketype.setIcon(emptyIcon);
457 
458         // Données numériques
459         lblTlx.setText(emptyString);
460         lblTly.setText(emptyString);
461         lblBrx.setText(emptyString);
462         lblBry.setText(emptyString);
463 
464         lblDx.setText(emptyString);
465         lblDy.setText(emptyString);
466 
467         lblCx.setText(emptyString);
468         lblCy.setText(emptyString);
469     }
470 }
```



```
1  package widgets;
2  
3  import java.awt.Component;
4  import java.awt.Dimension;
5  import java.awt.Font;
6  import java.awt.event.ItemListener;
7  
8  import javax.swing.BoxLayout;
9  import javax.swing.ImageIcon;
10 import javax.swing.JComboBox;
11 import javax.swing.JLabel;
12 import javax.swing.JList;
13 import javax.swing.JPanel;
14 import javax.swing.ListCellRenderer;
15 import javax.swing.SwingConstants;
16 
17 import utils.IconItem;
18 
19 /**
20  * Classe contenant un titre et une liste déroulante utilisant des JLabel avec
21  * des icones pour les élements de la liste déroulante
22  */
23 public class JLabeledComboBox extends JPanel
24 {
25     /** Le titre de cette liste */
26     private String title;
27 
28     /**
29      * Les textes et icônes pour les items
30      */
31     private IconItem[] items;
32 
33     /**
34      * La combobox utilisée à l'intérieur pour pouvoir ajouter des listener
35      * par la suite
36      */
37     private JComboBox<IconItem> combobox;
38 
39     /**
40      * Constructeur
41      * @param title le titre du panel
42      * @param captions les légendes des éléments de la liste
43      * @param selectedIndex l'élément sélectionné initialement
44      * @param listener le listener à appeller quand l'élement sélectionné de la
45      *            liste change
46      * @see #createImageIcon(String)
47      */
48     public JLabeledComboBox(String title, String[] captions, int selectedIndex,
49             ItemListener listener)
50     {
51         setAlignmentX(Component.LEFT_ALIGNMENT);
52 
53         this.title = title;
54         items = new IconItem[captions.length];
55 
56         for (int i = 0; i < captions.length; i++)
57         {
58             items[i] = new IconItem(captions[i]);
59         }
60 
61         setLayout(new BoxLayout(this, BoxLayout.X_AXIS));
62 
63         // Creates the title
64         JLabel label = new JLabel((this.title ≠ null ? this.title : "text"));
65         label.setHorizontalAlignment(SwingConstants.LEFT);
66         add(label);
67 
68         // Creates the Combobox
69         combobox = new JComboBox<IconItem>(items);
70         combobox.setAlignmentX(Component.LEFT_ALIGNMENT);
71         combobox.setEditable(false);
72         int index;
73         if ((selectedIndex < 0) ∨ (selectedIndex > captions.length))
74         {
75             index = 0;
76         }
77         else
78         {
79             index = selectedIndex;
80         }
81         combobox.setSelectedIndex(index);
82         combobox.addItemListener(listener);
83         // Mise en place du renderer pour les élements de la liste
84         JLabelRenderer renderer = new JLabelRenderer();
85         renderer.setPreferredSize(new Dimension(100, 32));
86         combobox.setRenderer(renderer);
87         // Ajout de la liste
88         add(combobox);
89     }
90 
```

```
91      /**
92       * Ajout d'un nouveau listener déclenché lorsqu'un élément est sélectionné
93       * @param aListener le nouveau listener à ajouter.
94       */
95      public void addItemListener(ItemListener aListener)
96      {
97          if (combobox ≠ null)
98          {
99              combobox.addItemListener(aListener);
100         }
101         else
102         {
103             System.err.println(getClass().getSimpleName() + "::addItemListener : null combobox");
104         }
105     }
106
107     /**
108      * Obtention de l'index de l'élément sélectionné dans le combobox
109      * @return l'index de l'élément sélectionné dans le combobox
110      */
111     public int getSelectedIndex()
112     {
113         return combobox.getSelectedIndex();
114     }
115
116     /**
117      * Renderer pour les Labels du combobox
118      */
119     protected class JLabelRenderer extends JLabel
120         implements ListCellRenderer<IconItem>
121     {
122         /** fonte pour les items à problèmes */
123         private Font pbFont;
124
125         /**
126          * Constructeur
127          */
128         public JLabelRenderer()
129         {
130             setOpaque(true);
131             setHorizontalAlignment(LEFT);
132             setVerticalAlignment(CENTER);
133         }
134
135         /*
136          * (non-Javadoc)
137          * @see
138          * javax.swing.ListCellRenderer#getListCellRendererComponent(javax.swing
139          * .JList, java.lang.Object, int, boolean, boolean)
140          */
141         @Override
142         public Component getListCellRendererComponent(
143             JList<? extends IconItem> list, IconItem value, int index,
144             boolean isSelected, boolean cellHasFocus)
145         {
146             if (isSelected)
147             {
148                 setBackground(list.getSelectionBackground());
149                 setForeground(list.getSelectionForeground());
150             }
151             else
152             {
153                 setBackground(list.getBackground());
154                 setForeground(list.getForeground());
155             }
156
157             // Mise en place de l'icone et du texte dans le label
158             // Si l'icone est null afficher un label particulier avec
159             // setPbText
160             ImageIcon itemIcon = value.getIcon();
161             String itemString = value.getCaption();
162             setIcon(itemIcon);
163             if (itemIcon ≠ null)
164             {
165                 setText(itemString);
166                 setFont(list.getFont());
167             }
168             else
169             {
170                 setPbText(itemString + " (pas d'image)", list.getFont());
171             }
172
173             return this;
174         }
175
176         /**
177          * Mise en place du texte s'il y a un pb pour cet item
178          * @param pbText le texte à afficher
179          * @param normalFont la fonte à utiliser (italique)
180          */
```



```
181         protected void setPbText(String pbText, Font normalFont)
182         {
183             if (pbFont ≡ null)
184             { // lazily create this font
185                 pbFont = normalFont.deriveFont(Font.ITALIC);
186             }
187             setFont(pbFont);
188             setText(pbText);
189         }
190     }
191 }
```

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



```java
package widgets;

import java.awt.BorderLayout;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.util.Observer;

import javax.swing.DefaultComboBoxModel;
import javax.swing.JComboBox;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.ScrollPaneConstants;
import javax.swing.tree.TreeModel;

import figures.Drawing;
import figures.treemodels.FigureTreeModel;
import widgets.enums.TreeType;

public class TreesPanel extends JPanel
{
    /**
     * Le type d'arbre que l'on veut utiliser
     * @see TreeType
     */
    private TreeType treeType;

    /**
     * Le modèle d'arbre à créer en fonction du {@link #treeType}
     */
    private TreeModel model;

    /**
     * Le modèle de dessin
     */
    private Drawing drawing;

    /**
     * Le {@link JTree} à utiliser pour visualiser l'arbre
     */
    private JTree tree;

    /**
     * Change le type d'arbre et crée le TreeModel associé
     * @param treeType the treeType to set
     */
    public void setTreeType(TreeType treeType)
    {
        System.out.println("setTreeType(" + treeType + ")");
        this.treeType = treeType;

        if (model ≠ null)
        {
            drawing.deleteObserver((Observer) model);
            model = null;
        }

        if ((drawing ≠ null) ∧ (tree ≠ null))
        {
            switch (this.treeType)
            {
                case FIGURE:
                    model = new FigureTreeModel(drawing, tree);
                    break;
                case FIGURE_TYPE:
                    model = null; // TODO new FigureTypeTreeModel(drawing, tree);
                    break;
                case FILL_COLOR:
                    model = null; // TODO new FillColorTreeModel(drawing, tree);
                    break;
                case EDGE_COLOR:
                    model = null; // TODO new EdgeColorTreeModel(drawing, tree);
                    break;
                case EDGE_TYPE:
                    model = null; // TODO new EdgeTypeTreeModel(drawing, tree);
                    break;
                default:
                    model = null;
                    break;
            }
        }
        else
        {
            System.out.println("FigureTypeTreeModel not set up because "
                + "null drawing or null JTree");
        }
    }

    /**
```

```
91      * Sets the drawing
92      * @param drawing the drawing to set
93      */
94     public void setDrawing(Drawing drawing)
95     {
96         // System.out.println("Setting up Drawing" + drawing + " in
97         // TreesPanel");
98         this.drawing = drawing;
99         if (drawing ≠ null)
100        {
101            setTreeType(treeType);
102        }
103        else
104        {
105            System.err.println(getClass().getSimpleName() + "::setDrawing : null drawing");
106        }
107    }
108
109    /**
110     * Create the panel.
111     */
112    public TreesPanel()
113    {
114        int treeTypeIndex = 0;
115        treeType = TreeType.fromInteger(treeTypeIndex);
116        model = null;
117        setLayout(new BorderLayout(0, 0));
118
119        JPanel treeModePanel = new JPanel();
120        add(treeModePanel, BorderLayout.NORTH);
121        treeModePanel.setLayout(new BorderLayout(0, 0));
122
123        JLabel lblTreeMode = new JLabel("Tree mode");
124        treeModePanel.add(lblTreeMode, BorderLayout.WEST);
125
126        JComboBox<TreeType> treeComboBox = new JComboBox<TreeType>();
127        treeComboBox.setMaximumRowCount(TreeType.NbTreeTypes);
128        treeComboBox
129            .setModel(new DefaultComboBoxModel<TreeType>(TreeType.values()));
130        treeComboBox.setSelectedIndex(treeTypeIndex);
131        treeComboBox.addItemListener(new ItemListener()
132        {
133            @Override
134            public void itemStateChanged(ItemEvent e)
135            {
136                @SuppressWarnings("unchecked")
137                JComboBox<TreeType> combo = (JComboBox<TreeType>) e.getSource();
138                if (e.getStateChange() ≡ ItemEvent.SELECTED)
139                {
140                    Object selectedItem = combo.getSelectedItem();
141                    if (selectedItem instanceof TreeType)
142                    {
143                        setTreeType((TreeType) selectedItem);
144
145                        System.out.println("Setting tree type to " +
146                        selectedItem);
147                    }
148                }
149            }
150        });
151        treeModePanel.add(treeComboBox);
152
153        JScrollPane treeScrollPane = new JScrollPane();
154        treeScrollPane.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);
155        add(treeScrollPane, BorderLayout.CENTER);
156
157        tree = new JTree();
158        treeScrollPane.setViewportView(tree);
159    }
160 }
```