```java
package maze.display;

import maze.data.MazeContainer;

/**
 * A class that displays a textual version of the maze given
 * in the form of a {@link MazeContainer}
 *
 * @author Pierre-Andre Mudry
 * @date February 29th, 2012
 * @version 1.0
 */
public class TextDisplay {

    /**
     * Displays the maze given in parameter on the default console
     *
     * @param mazeC The {@link MazeContainer} to display
     */
    public static void displayMaze(MazeContainer mazeC) {

        // Get the real labyrinth
        MazeElem[][] maze = mazeC.maze;

        // Size of the labyrinth
        int nCellsX = mazeC.nCellsX;
        int nCellsY = mazeC.nCellsY;

        /**
         * Draw the labyrinth
         */
        for (int i = 0; i < nCellsY; i++) {
            // Draws the north edge
            for (int j = 0; j < nCellsX; j++) {
                MazeElem e = maze[j][i];

                if (e.wallNorth)
                    System.out.print("*---");
                else
                    System.out.print("*   ");
            }

            System.out.println("*");

            // Draws the west edge
            for (int j = 0; j < nCellsX; j++) {
                MazeElem e = maze[j][i];
                if (e.wallWest) {
                    if (e.p1Present)
                        System.out.print("| p1");
                    else if (e.p2Present)
                        System.out.print("| p2");
                    else if (e.isExit)
                        System.out.print("| e ");
                    else
                        System.out.print("|   ");
                } else {
                    if (e.p1Present)
                        System.out.print("  p1");
                    else if (e.p2Present)
                        System.out.print("  p2");
                    else if (e.isExit)
                        System.out.print("  e ");
                    else
                        System.out.print("    ");
                }

            }
            System.out.println("|");
        }
        // Draws the bottom line
        for (int j = 0; j < nCellsX; j++) {
            System.out.print("*---");
        }
        System.out.println("*");
    }

    public static void main(String args[]){
        MazeContainer mg = new MazeContainer(5, 5);
        TextDisplay.displayMaze(mg);
    }
}

```

```java
package maze.display;

import hevs.graphics.ImageGraphicsMultiBuffer;

/**
 * A graphic view of a {@link MazeContainer}
 *
 * @author Pierre-Andre Mudry
 * @version 1.5
 * @date February 2012
 */
public class GraphicDisplay {

    // The number of cells
    public final int nCellsX, nCellsY;

    /**
     * Window and drawing related
     */
    // Dimensions (in pixels) of each cell
    public final int wCell;
    public final int hCell;

    // Size of the whole screen
    public final int frameWidth, frameHeight;

    // Shall we draw the grid ?
    boolean drawGrid = false;

    // Size of the stroke (grid and maze)
    private int strokeSize = 7;

    /**
     * UI related
     */
    // The logo
    private BufferedImage mBitmap;

    // The message at the bottom of the screen
    private String msg;

    // Contains the maze that we will display
    private MazeContainer mazeContainer;

    // Contains the Display that is used to show the maze
    public Display disp;

    int[][] solution;

    /**
     * FIXME
     * @param kl
     */
    public void registerKeyListener(KeyboardListener kl) {
        disp.registerKeyListener(kl);
    }

    /**
     * Sets the message that will be displayed at the bottom of the screen
     *
     * @param msg
     */
    public void setMessage(String msg) {
        disp.setMessage(msg);
    }

    /**
     * Sets a new maze for display
     *
     * @param mc
     */
    public void setNewMaze(MazeContainer mc) {
        this.mazeContainer = mc;
    }

    public class Display extends ImageGraphicsMultiBuffer {
        private static final long serialVersionUID = 1L;

        public Display(String title, int width, int height, boolean hasDecoration) {
            super(title, width, height, hasDecoration);
        }

        public void registerKeyListener(KeyListener kl) {
            super.mainFrame.addKeyListener(kl);
        }


```

```
102         /**
103          * Sets the text that is displayed at the bottom of the screen
104          *
105          * @param msg
106          */
107         public void setMessage String message  {
108             msg = message;
109         }
110 
111         /**
112          * Does the rendering process for the maze
113          */
114         @Override
115         public void render Graphics2D g  {
116 
117             /**
118              * Take the borders into account if we are rendering with Swing
119              * decoration
120              */
121             int border_top = this.mainFrame.getInsets().top;
122             int border_left = this.mainFrame.getInsets().left;
123 
124             int xs = border_left + strokeSize / 2, ys = border_top + strokeSize / 2;
125 
126             // Set the pen size using the stroke
127             g.setStroke new BasicStroke strokeSize );
128 
129             /**
130              * Grid drawing
131              */
132             if drawGrid  {
133                 g.setColor new Color 220, 220, 220 );
134 
135                 // Horizontal grid lines
136                 for  int i = 0; i < nCellsY + 1; i++)  {
137                     g.drawLine 0, ys, frameWidth - strokeSize + border_top, ys );
138                     ys += hCell + strokeSize;
139                 }
140 
141                 // Vertical grid lines
142                 for  int i = 0; i < nCellsX + 1; i++)  {
143                     g.drawLine xs, 0, xs, frameHeight - strokeSize + border_top );
144                     xs += wCell + strokeSize;
145                 }
146             }
147 
148             /**
149              * Draw the content of the maze
150              */
151             g.setColor Color.BLACK );
152             xs = border_left + strokeSize / 2;
153             ys = border_top + strokeSize / 2;
154 
155             // Draw the solution if required
156             if solution != null  {
157                 for  int i = 0; i < nCellsX; i++)  {
158                     for  int j = 0; j < nCellsY; j++)  {
159                         MazeElem e = mazeContainer.maze[i][j];
160 
161                         // Draw solution
162                         if solution != null && solution[i][j] == 1  {
163                             g.setColor new Color 200, 200, 250 );
164                             g.fillRect xs, ys, wCell + strokeSize, hCell + strokeSize );
165                             g.setColor Color.black );
166                         }
167                         ys += hCell + strokeSize;
168                     }
169 
170                     ys = border_top + strokeSize / 2;
171                     xs += wCell + strokeSize;
172                 }
173             }
174 
175             xs = border_left + strokeSize / 2;
176             ys = border_top + strokeSize / 2;
177 
178             // Draw the content of the frames
179             for  int i = 0; i < nCellsX; i++)  {
180                 // draw the north edge
181                 for  int j = 0; j < nCellsY; j++)  {
182                     MazeElem e = mazeContainer.maze[i][j];
183 
184                     // Draw exit
185                     if e.isExit  {
186                         g.setColor new Color 100, 100, 200 );
187                         g.fillRect
```

```
188                             xs + (int) Math.round(strokeSize / 2.0), ys + (int) Math.round(strokeSize / 2.0),
189                             wCell, hCell);
190                     g.setColor(Color.black);
191                 }
192
193                 // Draw position for player 1
194                 if (e.p1Present) {
195                     g.setColor(Color.red);
196                     g.fillOval(
197                             xs + (int) Math.round(strokeSize / 2.0), ys + (int) Math.round(strokeSize / 2.0),
198                             wCell, hCell);
199                     g.setColor(Color.black);
200                     g.setStroke(new BasicStroke(1.0f));
201                     g.drawOval(
202                             xs + (int) Math.round(strokeSize / 2.0), ys + (int) Math.round(strokeSize / 2.0),
203                             wCell, hCell);
204                     g.setStroke(new BasicStroke(strokeSize));
205                 }
206
207                 if (e.p2Present) {
208                     g.setColor(Color.yellow);
209                     g.fillOval(
210                             xs + (int) Math.round(strokeSize / 2.0), ys + (int) Math.round(strokeSize / 2.0),
211                             wCell, hCell);
212                     g.setColor(Color.black);
213                     g.setStroke(new BasicStroke(1.0f));
214                     g.drawOval(
215                             xs + (int) Math.round(strokeSize / 2.0), ys + (int) Math.round(strokeSize / 2.0),
216                             wCell, hCell);
217                     g.setStroke(new BasicStroke(strokeSize));
218                 }
219
220                 // Is there a north wall ?
221                 if (e.wallNorth) {
222                     g.drawLine(xs, ys, xs + wCell + strokeSize, ys);
223                 }
224
225                 // Is there a left wall ?
226                 if (e.wallWest) {
227                     g.drawLine(xs, ys, xs, ys + hCell + strokeSize);
228                 }
229
230                 // Draw bottom for the last line
231                 if ((j == nCellsY - 1) && (e.wallSouth)) {
232                     g.drawLine(xs, ys + hCell + strokeSize, xs + wCell + strokeSize, ys + hCell + strokeSize);
233                 }
234
235                 // Draw right for the last column
236                 if ((i == nCellsX - 1) && (e.wallEast)) {
237                     g.drawLine(xs + wCell + strokeSize, ys, xs + wCell + strokeSize, ys + hCell + strokeSize);
238                 }
239
240                 ys += hCell + strokeSize;
241             }
242
243             ys = border_top + strokeSize / 2;
244             xs += wCell + strokeSize;
245         }
246
247         /**
248          * Draw HES-SO logo, centered, at the bottom of the screen
249          */
250         g.drawImage(mBitmap, fWidth / 2 - mBitmap.getWidth() / 2, fHeight - 30, null);
251
252         // Write some information message
253         if (msg != null)
254             g.drawString(msg, 5, fHeight - 40);
255     }
256 }
257
258 /**
259  * This method is used to overlay a solution that has been found using one
260  * solver algorithm such as the one implemented in {@link AStar_BEGIN}
261  *
262  * @param solution The solution to overlay
263  */
264 public void setSolution(int[][] solution) {
265     assert(solution.length == nCellsX);
266     assert(solution[0].length == nCellsY);
267     this.solution = solution;
268 }
269
270 /**
271  * Call this method to remove the solution overlay
272  */
273 public void clearSolution() {
```

```
274         this.solution = null;
275     }
276 
277     /**
278      * Loads an image into mBitmap
279      *
280      * @param imageName The image path to be loaded (relative to the src/bin
281      *            folder), i.e. /images/...)
282      */
283     private void loadImage String imageName) {
284         try {
285             mBitmap = ImageIO.read SimpleGraphicsBitmap.class.getResource imageName));
286 
287         } catch (Exception e) {
288             System.out.println("Could not find image " + imageName + ", exiting !");
289             e.printStackTrace();
290             System.exit(-1);
291         }
292     }
293 
294     /**
295      * @see GraphicDisplay
296      */
297     public GraphicDisplay MazeContainer mc, int sizeOfSquare) {
298         this(mc, sizeOfSquare, true);
299     }
300 
301     /**
302      * Display a window showing a {@link MazeContainer}
303      *
304      * @param mc The maze to show
305      * @param sizeOfSquare The width of each square to show
306      * @param decorations If we need the borders or not
307      */
308     public GraphicDisplay MazeContainer mc, int sizeOfSquare, boolean decorations) {
309         mazeContainer = mc;
310 
311         nCellsX = mc.nCellsX;
312         nCellsY = mc.nCellsY;
313 
314         /**
315          * Compute the sizes for the graphical display
316          */
317         wCell = sizeOfSquare;
318         hCell = sizeOfSquare;
319 
320         /**
321          * Size of the frame should have space for all the cells (nCellsX *
322          * wCell) and also space for the grid (hence the nCellsX + 1 *
323          * strokeWidth)
324          */
325         frameWidth = (nCellsX * wCell + ((nCellsX + 1) * strokeSize));
326         frameHeight = (nCellsY * hCell + ((nCellsY + 1) * strokeSize));
327 
328         // Load the image
329         loadImage("/images/logo_hei.png");
330 
331         // Create a display and keep some space for the picture and the text at
332         // the bottom
333         disp = new Display("Maze - Minilabor", frameWidth, frameHeight + 55, decorations);
334 
335         // Sets the default message
336         disp.setMessage("Welcome to the Maze game !");
337     }
338 
339     public static void main String args[]) {
340         // Generate a maze
341         MazeContainer mc = new MazeContainer(20, 15);
342 
343         // Display the maze
344         GraphicDisplay gd = new GraphicDisplay(mc, 15, true);
345     }
346 }
347 
```

```java
package maze.solvers;

import java.text.DecimalFormat;

/**
 * A-Star (Lee) algorithm for maze solving
 *
 * @author Pierre-Andre Mudry, Romain Cherix
 * @date February 2012
 * @version 1.2
 *
 */
public class AStar {

    private MazeElem[][] maze;
    private int width, height;
    private int[][] solution;

    // Debug information
    public final boolean VERBOSE = true;

    private AStar(MazeContainer mazeContainer) {
        maze = mazeContainer.maze;
        width = mazeContainer.nCellsX;
        height = mazeContainer.nCellsY;
    }

    /**
     * Solves the maze
     *
     * @param x The x-coordinate of the start point
     * @param y The y-coordinate of the start point
     */
    private void solve(int x, int y) {
        /**
         * The solution at the beginning is an array full of zeroes
         */
        solution = new int[width][height];

        // We indicate the starting position
        solution[x][y] = 1;

        // This is the step counter
        int m = 1;

        /**
         * Do the expansion until we have reached the exit.
         */
        while (expansion(m) == false) {
            m++;
        }

        /**
         * m contains the total number of steps to find the solution
         */
        if (VERBOSE)
            System.out.println("\n[AStar solver] Took " + m + " steps for the solution\n");

        /**
         * As the forward propagation is over, we can now do the back-prop
         * phase.
         */
        backtrace(m);
    }

    /**
     * Lee forward propagation algorithm
     *
     * @param m The current step of the algorithm
     * @return A boolean value that indicates if wave has hit exit
     */
    private boolean expansion(int m) {

        for (int j = 0; j < height; j++) {
            for (int i = 0; i < width; i++) {

                /**
                 * At each step m, we propagate the wave for each cell of the
                 * solution that has the index m.
                 */
                if (solution[i][j] == m) {
                    if (!maze[i][j].wallWest)
                        if (maze[i][j].isExit) {
                            return true;
                        } else if (solution[i - 1][j] == 0)
                            solution[i - 1][j] = m + 1;
```

```
92
93                      if (!maze[i][j].wallNorth)
94                          if (maze[i][j].isExit) {
95                              return true;
96                          } else if (solution[i][j - 1] == 0)
97                              solution[i][j - 1] = m + 1;
98
99                      if (!maze[i][j].wallEast)
100                         if (maze[i][j].isExit) {
101                             return true;
102                         } else if (solution[i + 1][j] == 0)
103                             solution[i + 1][j] = m + 1;
104
105                     if (!maze[i][j].wallSouth)
106                         if (maze[i][j].isExit) {
107                             return true;
108                         } else if (solution[i][j + 1] == 0)
109                             solution[i][j + 1] = m + 1;
110                 }
111             }
112         }
113
114         return false;
115     }
116
117     /**
118      * Grants uniform access for the whole maze and makes sure that we do not
119      * cross the borders of the maze
120      *
121      * @param i x position
122      * @param j y position
123      * @return distance to the origin point, -1 if outside the graph
124      */
125     private int access_solution(int i, int j) {
126         if (i >= width || i < 0 || j >= height || j < 0)
127             return -1;
128         else
129             return solution[i][j];
130     }
131
132     /**
133      * Lee algorithm back-trace phase when the array has been annotated with the
134      * distances
135      *
136      * @param m The highest distance from origin point
137      */
138     private void backtrace(int m) {
139         int[][] ret = new int[width][height];
140
141         int x = 0, y = 0;
142
143         // Get the coordinates of exit in original maze
144         for (int j = 0; j < height; j++) {
145             for (int i = 0; i < width; i++) {
146                 if (maze[i][j].isExit) {
147                     x = i;
148                     y = j;
149                     break;
150                 }
151             }
152         }
153
154         // The exit is part of the solution
155         ret[x][y] = 1;
156
157         /**
158          * While we haven't reached the beginning, annotate the solution with
159          * the correct path
160          */
161         while (m > 0) {
162             if (access_solution(x - 1, y) == m && !maze[x][y].wallWest)
163                 ret[--x][y] = 1;
164
165             if (access_solution(x, y - 1) == m && !maze[x][y].wallNorth)
166                 ret[x][--y] = 1;
167
168             if (access_solution(x + 1, y) == m && !maze[x][y].wallEast)
169                 ret[++x][y] = 1;
170
171             if (access_solution(x, y + 1) == m && !maze[x][y].wallSouth)
172                 ret[x][++y] = 1;
173
174             m--;
175         }
176
177         // Update the solution with the backprop version
```

```java
        solution = ret;
    }

    /**
     * Displays the solution on the console for control
     */
    public static void displaySolution(int[][] mazeSolution) {
        String solutionText = "";

        int width = mazeSolution[0].length;
        int height = mazeSolution.length;

        if (mazeSolution != null)
            for (int j = 0; j < width; j++) {
                for  int i = 0; i < height; i++) {

                    DecimalFormat myFormatter = new DecimalFormat("00");
                    String s = myFormatter.format(mazeSolution[i][j]);

                    if (i != height - 1)
                        solutionText += s + " - ";
                    else
                        solutionText += s;
                }
                solutionText += "\n";
            }

        System.out.println(solutionText);
    }

    /**
     * This class is thought to be used statically using only this method
     *
     * @param mc The {@link MazeContainer} that we want to solve
     * @param x The x-coordinate of the start point
     * @param y The y-coordinate of the start point
     * @return An array containing 1's along the solution path
     */
    public static int[][] solve(MazeContainer mc, int x, int y) {
        AStar alg = new AStar(mc);
        alg.solve(x, y);
        return alg.solution;
    }

    public static void main(String args[]) {
        /**
         * Create a maze and display its textual representation
         */
        MazeContainer mc = new MazeContainer(50, 80);
        TextDisplay.displayMaze(mc);

        /**
         * Compute a solution and display it
         */
        int[][] solution = AStar.solve(mc, 0, 0);
        AStar.displaySolution(solution);

        GraphicDisplay gd = new GraphicDisplay(mc, 2, false);
        gd.setSolution(solution);

    }

}
```

```java
package maze;

import java.awt.Point;

/**
 * Game logic for moving the players and selectively showing the solution
 *
 * @author Pierre-Andre Mudry, Romain Cherix
 * @date February 2012
 * @version 1.4
 */
public class MazeGame {

    public MazeElem[][] maze;
    int width, height;

    GraphicDisplay gd;
    MazeContainer mc;
    KeyboardListener kl;

    /**
     * By default, you are the first player but this can change..
     */
    Player player = Player.PLAYER1;

    MazeGame MazeContainer mc) {
        gd = new GraphicDisplay(mc, 5);

        // Link key presses with the actions
        kl = new KeyboardListener mc, this);
        gd.registerKeyListener(kl);

        setNewMaze(mc);
    }

    /**
     * Dynamically changes the maze that is displayed
     *
     * @param mc
     */
    protected void setNewMaze MazeContainer mc) {
        maze = mc.maze;
        width = mc.nCellsX;
        height = mc.nCellsY;
        this.mc = mc;

        /**
         * Update graphical maze
         * FIXME : this should allow mazes of different sizes,
         * which is not the case now
         */
        gd.setNewMaze(mc);
    }

    /**
     * Call this when you want a new game
     * @param mazeID
     */
    public void generateNewMaze(int mazeID){
        this.setNewMaze new MazeContainer(width, height, mazeID));
    }

    /**
     * Displays the solution on the screen for a player during a whole second
     *
     * @param p Which player's solution do you want
     */
    public void displaySolution() {
        Point p1 = findPlayer(player);

        int[][] solution = AStar.solve(mc, p1.x, p1.y);
        gd.setSolution(solution);

        Timer timer = new Timer(1000, new ActionListener() {
            public void actionPerformed ActionEvent e) {
                gd.clearSolution();
            }
        });

        timer.setRepeats(false);
        timer.start();
    }

    /**
     * Gives us the location of player inside the maze
     *
```

```
101      * @param p The player we want
102      * @return The location of the player
103      */
104     private Point findPlayer(Player p) {
105         /**
106          * We go through all the elements
107          */
108         for (int j = 0; j < height; j++) {
109             for (int i = 0; i < width; i++) {
110                 MazeElem e = maze[i][j];
111 
112                 // Once found, get it back
113                 if (p == Player.PLAYER1 && e.p1Present)
114                     return new Point(i, j);
115 
116                 if (p == Player.PLAYER2 && e.p2Present)
117                     return new Point(i, j);
118             }
119         }
120 
121         // This means that the player hasn't been found
122         // which can happen for instance in single player games
123         return null;
124     }
125 
126     /**
127      * Check if some player has reached the exit of the maze
128      */
129     public boolean checkWinner() {
130         for (int j = 0; j < height; j++) {
131             for (int i = 0; i < width; i++) {
132                 MazeElem el = maze[i][j];
133 
134                 if (el.isExit && el.p1Present) {
135                     JOptionPane.showMessageDialog(
136                             null, "You won !", "We have a winner !", JOptionPane.INFORMATION_MESSAGE);
137                     return true;
138                 }
139             }
140         }
141         return false;
142     }
143 
144     /**
145      * Method used to move a player inside the maze
146      *
147      * @param d Which direction do you want to go to ?
148      */
149     public void movePlayer(Direction d) {
150         boolean movementDone = false;
151 
152         for (int j = 0; j < height; j++){
153             for (int i = 0; i < width; i++) {
154 
155                 MazeElem e = maze[i][j];
156 
157                 if (e.p1Present && player == Player.PLAYER1) {
158                     movementDone = true;
159 
160                     switch (d) {
161                     case UP:
162                         if (!e.wallNorth) {
163                             e.p1Present = false;
164                             maze[i][j - 1].p1Present = true;
165                         }
166                         break;
167 
168                     case DOWN:
169                         if (!e.wallSouth) {
170                             e.p1Present = false;
171                             maze[i][j + 1].p1Present = true;
172                         }
173                         break;
174 
175                     case RIGHT:
176                         if (!e.wallEast) {
177                             e.p1Present = false;
178                             maze[i + 1][j].p1Present = true;
179                         }
180                         break;
181 
182                     case LEFT:
183                         if (!e.wallWest) {
184                             e.p1Present = false;
185                             maze[i - 1][j].p1Present = true;
186                         }
```

```
187                         break;
188                     }
189
190                 }
191
192                 if(movementDone) break;
193             }
194             if(movementDone) break;
195         }
196
197         /**
198          * When the move has been done, see if there is a winner
199          */
200         if(checkWinner())
201             generateNewMaze(new Random().nextInt());
202     }
203
204     /**
205      * Only for fun, generate hundreds of labyrinths per second
206      */
207     public void multiShuffle()
208     {
209         Timer timer = new Timer(100, new ActionListener() {
210             public void actionPerformed(ActionEvent e) {
211                 Random rnd = new Random();
212                 mc = new MazeContainer(15, 15, rnd.nextInt());
213                 gd.setNewMaze(mc);
214             }
215         });
216
217         timer.setRepeats(true);
218         timer.start();
219     }
220
221     public static void main(String[] args) {
222         MazeContainer mc = new MazeContainer(100, 50);
223         MazeGame mg = new MazeGame(mc);
224
225 //      mg.multiShuffle();
226
227         // TODO Students should implement next line
228         mg.movePlayer(Direction.DOWN);
229
230         // This shows a nice message window
231         // JOptionPane.showMessageDialog(null, "Title of the window", "Text of the window !",
    JOptionPane.INFORMATION_MESSAGE);
232     }
233 }
234
```

```java
package maze.display;

import java.awt.event.KeyEvent;

/**
 * Links key presses and actions
 *
 * @author Pierre-Andre Mudry
 * @date February 2012
 * @version 1.0
 */
public class KeyboardListener implements KeyListener {

    MazeGame mg;
    MazeContainer mc;

    /**
     * To link keys to actions from the game in the maze, we need references to
     * both
     *
     * @param mc The maze
     * @param mg The game
     */
    public KeyboardListener MazeContainer mc, MazeGame mg  {
        this.mc = mc;
        this.mg = mg;
    }

    /**
     * What happens when a key has been pressed
     */
    @Override
    public void keyPressed KeyEvent arg0  {

        /**
         * Keys for player 1
         */
        switch  arg0.getKeyCode()) {
        case KeyEvent.VK_W:
            mg.movePlayer(Direction.UP);
            break;
        case KeyEvent.VK_S:
            mg.movePlayer(Direction.DOWN );
            break;
        case KeyEvent.VK_D:
            mg.movePlayer(Direction.RIGHT);
            break;
        case KeyEvent.VK_A:
            mg.movePlayer(Direction.LEFT );
            break;
        case KeyEvent.VK_Q:
            mg.displaySolution();
            break;
        case KeyEvent.VK_N:
            mg.generateNewMaze(new Random().nextInt());
            break;
        }
    }

    /**
     * This method is called when a key has been released (i.e. no more pressed)
     */
    @Override
    public void keyReleased KeyEvent arg0  {

    }

    /**
     * This method is called when a key has been pressed and released (complete
     * cycle)
     */
    @Override
    public void keyTyped KeyEvent arg0) {

    }
}

```