

Cómo pintar tableros en Swing

Así que quieres pintar un tablero. Asumamos que dispones de una clase `Board` con los siguientes métodos:

```
interface Board {
   Piece getPosition(int row, int col);
   int getRows();
   int getCols();
   // ...
}
```

y, ya puestos, algo que te diga de qué color pintar las fichas:

```
interface PieceColorMap {
   Color getColofFor(Piece p);
}
```

el primer objetivo es llegar a algo cuya interfaz externa sea similar a

```
public class BoardUI extends JPanel {
   public BoardUI();
   public void setBoard(Board board); // totally replace the board
   public void setColors(PieceColorMap colorMap);
   public void update();        // repaint the board; rows and columns have not changed
}
```

También puedes hacer que `setBoard()` y `update()` sean privados, y registrarte de alguna forma a los eventos del juego vía `GameObserver`:

```
public class BoardUI extends JPanel {
   public BoardUI();
   public void setGame(Game game); // registers for GameObserver events
   public void setColors(PieceColorMap colorMap);
}
```

pero esto te obliga a hacer algo similar a lo siguiente:

```
private Game game;
private GameObserver observer = new GameObserver() {
   public void onGameStart(
       Board board, String gameDesc, List<Piece> pieces, Piece turn) {     setBoard(board); }
   public void onGameOver(Board board, Game.State state, Piece winner) { update(); }
   public void onMoveEnd(Board board, Piece turn, boolean success) { update(); }
   public void onChangeTurn(Board board, Piece turn) { update(); }
```

```
   public void onError(String msg) { /* ignore */ }
};
public void setGame(Game game) {
   if (this.game != null) game.removeObserver(observer);
   this.game = game;
   game.addObserver(observer);
}
```

Que, al menos inicialmente, es más complicado de probar. Tarde o temprano deberías implementarlo, pero sólo una vez hayas verificado que puedes pintar bien tableros y generar bien movimientos.

Concentrémonos en cómo pintas el tablero, asumiendo que lo tienes a mano. Hay dos vías principales, y ambas son invisibles a un usuario externo de la clase (a quien, en tanto en cuanto puedas pintar tableros, le da igual cómo lo haces):

La manera fácil

Usaremos un montón de `JLabels` en un `GridLayout`. Para la generación de movimientos, querremos que los `JLabels` sepan su propia posición (= usaremos una subclase con campos para guardar esa información, pasada a través del constructor); pero, por ahora, vamos a concentrarnos en pintar cosas:

```
private JLabel[][] squares;
private Board board;

public void setBoard(Board board) {
   removeAll(); // descartamos squares antiguos
   this.board = board;
   squares = new Jlabel[board.getRows()][board.getColumns()];
   setLayout(new GridLayout(board.getRows(), board.getColumns());
   for (int i=0; i<board.getRows()) {
       for (int j=0; j<board.getColumns()) {
           squares[i][j] = new JLabel();
       }
   }
}
public void update() {
   for (int i=0; i<board.getRows()) {
       for (int j=0; j<board.getColumns()) {
           Piece p = board.getPosition(i, j);
           // y aquí configuras squares[i][j] para que pinte algo que represente 'p'
       }
   }
   repaint(); // obligas a que se repinte el tablero (ahora que pintará lo que debe)
}
```

Es decir – todo el pintado queda delegado a los `Jlabels`, que se colocan por sí mismos en sus

posiciones.

La forma difícil

También puedes escribirte un `JPanel` que sepa cómo pintar los cuadrados echando cuentas él solito, sin necesidad de `JLabel` alguno. Esto no es recomendable (escribir código innecesario es malo para la salud; y además, los redondeos pueden causar algún artefacto visual, porque quedarán píxeles sin usar cuando las divisiones no sean exactas), pero sí puede ser educativo:

```
private Board board;

public void setBoard(Board board) {
   this.board = board;
}
public void paintComponent(Graphics g) {
   int h = getHeight() / board.getRows();
   int w = getWidth() / board.getColumns();
   for (int i=0; i<board.getRows()) {
       for (int j=0; j<board.getColumns()) {
           paintPiece(g, j*w, i*h, w, h, board.getPosition(i, j));
       }
   }
}
public void paintPiece(Graphics g, int x, int y, int width, int height, Piece p) {
   // usa g.fillRect() o similar para pintar la pieza p en esa posición y con esos tamaños
}
public void update() {
   repaint();
}
```

En esta segunda versión, aunque el código puede quedar más corto, hay un coste oculto: hay que escribir también código para saber en qué celda se ha pulsado (no es que sea un código complicado; pero sí hay que hacer algo de aritmética modular dentro e un `MouseListener`).

Generando movimientos

Cuando quieras generar movimientos, necesitarás más atributos. Como poco, querrás saber para qué ficha vas a generar movimientos, y a quién llamar una vez generados para validarlos y llevarlos a cabo.

Asumiendo que ya tienes estos atributos a mano, generar movimientos es tan sencillo como llamar a un método como el siguiente cada vez que recibes un click:

```
private void squareWasClicked(int row, int col) {
   // gestiona un click en una celda
}
```

con parámetros adecuados. Esto es fácil de hacer en el caso fácil: basta con que tus `JLabels` sean un poco más listos y se acuerden de la posición en la que fueron metidos (en el interior de

`setBoard`, en lugar de `new JLabel()`, tendrás un `new Square(i, j)`):

```
public class Square extends JLabel {
   private int row, col;
   public Square(int row, int col) {
       this.row = row;
       this.col = col;
       addMouseListener(new MouseAdapter() {
           public void mouseClicked(MouseEvent e) {
               squareWasClicked(Square.this.row, Square.this.col);
           }
       });
   }
}
```

Y resulta ligeramente más difícil (no en líneas de código – pero tienes que tener la idea clara) en la versión de pintado manual:

```
// en el constructor de tu BoardUI
addMouseListener(new MouseAdapter() {
   public void mouseClicked(MouseEvent e) {
       if (board == null) return;
       int row = e.getPoint().y / (getHeight() / board.getRows());
       int col = e.getPoint().x / (getWidth() / board.getColumns());
       squareWasClicked(row, col);
   }
}
```

El interior de `squareWasClicked` decidiría si generar o no una jugada según un sencillo diagrama de estados, algo similar al siguiente pseudocódigo (al menos para el Ataxx; similar para el 3-en-raya avanzado):

```
si seleccionas pieza tuya,
   cambias la selección a la nueva pieza
si seleccionas casilla vacía,
   y tenías una pieza tuya ya seleccionada
       intentas mover esa pieza a esa casilla vacía, y si lo consigues, pierdes tu selección
```

Evidentemente, necesitas mantener la posición (fila y columna) de la pieza seleccionada como atributo de tu `BoardUI`.

En otros juegos, esta máquina de estados será distinta – y no necesitarás mantener una selección.

Sacando factor común

Lo único que cambia para los 4 juegos que se pueden jugar (Conecta-N, 3-en-raya, 3-en-raya avanzado y Ataxx) es la lógica para generar la jugada y verificar o no su validez antes de enviarla al tablero. Por tanto, es perfectamente factible (y recomendable) hacer algún tipo de `BoardUI` genérico, y tener versiones especializadas en cada juego.