Universidad Nacional de General Sarmiento
Instituto de Industria

Trabajo practico n°1

Programación III

Comisión: 2.

Docentes: Fernando Torres, Alejandro Nelis.

Alumnos: Sandoval Maximiliano, Romano Facundo, Franco Chamorro, Montenegro Tomas.

En el siguiente informe, nosotros vamos a explicar, acerca de la composición y organización de cómo se realizó el primer trabajo practico de la materia Programación III, en el cual implementamos diferentes y nuevas tecnológicas que tuvimos que aprender a utilizar (Como por ejemplo Windowsbuilder) para poder cumplir con los requerimientos pedidos por parte del enunciado de dicho trabajo practico.

En primera instancia comenzamos a trabajar sobre la clase "interfaz" sin tener en consideración el código de negocio. Para poder reestructurar el código tuvimos que reescribir parte del mismo para poder organizar las clases y de esa forma proporcionarle efectividad al proyecto.

Agregamos 3 packages distintos para organizar las clases:

En el package “Multimedia” creamos la clase “Sonidos” en donde guarda los métodos que ejecutan la música que se reproduce dentro de la interfaz y la que se genera cuando se pulsan los diferentes botones que están dentro de la misma.

```
public static void sonidoBoton()
      { try {
           Clip sonido = AudioSystem.getClip();
           File a = new File("src/multimedia/sonidoBoton.wav");
sonido.open(AudioSystem.getAudioInputStream(a));
           sonido.start();
         } catch (Exception tipoError) {
            System.out.println("" + tipoError);
         }
      }
```

En el código podemos observar que el método posee “try” y “catch” de manera obligatoria por sintaxis, ya que es muy propenso a fallar en caso de que no encuentre los archivos de audio por la ruta especificada, que, en este caso, se encuentran dentro del package.

En el package “Negocio” creamos la clase "Lógica" donde colocamos el código de negocio, en donde se encuentran los métodos cuyo principal objetivo es realizar operaciones matemáticas relacionadas a la suma de los valores que se encuentran los textFields.

```
      public static void calcularSumaColu(int numColu, int sumar)
            { if (columnas.containsKey(numColu)) { int sumaAuxColu
            = columnas.get(numColu);
            columnas.put(numColu, sumaAuxColu + sumar);
            } else columnas.put(numColu,
                  sumar);
      }
```

Este método ira sumando los valores de los textFields por columna y agregara el resultado en un HashMap que creamos para para guardar dicha operación para después comprobar la suma final.

Más precisamente, dicho método será utilizado en la clase “interfaz” por el método `almacenarSumas()` que ira agregando la sumas de las filas y las columnas al Map, como mencionamos anteriormente.

```
      private void almacenarSumas()
            { int numFila = 0; int
            numColu = 1;

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

                        if (j % constanteEsquina == 0) {
                              numFila++;
                              Logica.getFilas().put(numFila,
Integer.parseInt(cuadrilla[i][j].getName()));

                              numColu = 1;
                              Logica.calcularSumaColu(numColu,
Integer.parseInt(cuadrilla[i][j].getName()));

                        } else {

                              Logica.calcularSumaFila(numFila,
Integer.parseInt(cuadrilla[i][j].getName()));

                              numColu++;
                              Logica.calcularSumaColu(numColu,
Integer.parseInt(cuadrilla[i][j].getName()));
                        }
```

En el package “Presentación” creamos las clases “Main_TP1” e "Interfaz" donde modificamos y organizamos los métodos anteriormente creados. En un principio, sobre la clase “Interfaz”, habíamos realizado una cuadrilla de forma manual, es decir, colocando cada textField individual sobre la misma, pero como esto generaba una cuantiosa repetición de código, decidimos crear un método alternativo para optimizar la operación y así también poder reducir código que aporte en dicha organización.

El método creado se llama `dibujarCuadrilla()` cuyo objetivo es el de crear una matriz de textFields (cuadrilla), que mediante un ciclo “for”(en este caso son 2 porque es una matriz que maneja 2 índices), ira posicionando dichos textFields dentro de un radio simétrico definido por posiciones en “X” e “Y “ sobre el eje cartesiano de la interfaz.

```
for (int i = 0; i < constanteEsquina; i++) {
	for (int j = 0; j < constanteEsquina; j++) {
			int numeroRandom = codigo.crear_n_Random();


			cuadrilla[i][j] = new JTextField();


	if (j % constanteEsquina == 0) { // pega el salto cuando llega al brode
			posX = 45;// resetea pos de x posY += 45;//
			avanza en y cuadrilla[i][j].setBounds(posX,
			posY, 40, 40);
			cuadrilla[i][j].setName("" + numeroRandom);
	}
	else
	{
			posX += 45;// desplaza hacia la der
			cuadrilla[i][j].setBounds(posX, posY, 40, 40);
			cuadrilla[i][j].setName("" + numeroRandom);
```

Como podemos ver en el código, se irán seteando los textFields sobre la matriz `cuadrilla` que serán posicionados en función a los valores que vaya tomando el ciclo for.

Cada textField situados en la cuadrilla solo podrán admitir un único carácter numérico, que vaya del 1 al 9. De esta manera podremos limitar el ingreso de caracteres no validos o que no estén relacionados con la suma a la que se debe llegar. Con esto podemos evitar que se genere un error o cualquier otro evento no deseado. Además permite que el juego tenga una resolución valida, y se pueda mostrar al final una posible solución al juegador.

```
private void limitarInputUsuario() {
```

```
for (int cont = 0; cont < constanteEsquina; cont++) {
      int a = cont;
      for (int cont2 = 0; cont2 < constanteEsquina; cont2++) {
            int b = cont2;
            cuadrilla[a][b].addKeyListener(new KeyAdapter()
                  { public void keyTyped(KeyEvent e) { if
                  (time.isRunning()) { char caracter =
                  e.getKeyChar();

                              if (Logica.esUnNumuero(caracter)) {
                                    e.consume();
                              } else { if
                                    (cuadrilla[a]
                        [b].getText().length() >= 1)
                                          e.consume();
                              }
                        } else
                              e.consume();
                  }
```

La función principal del método anterior es limitar al usuario para que, dentro de los textFields de la cuadrilla, solo pueda colocar números de un solo digito, es decir, la cuadrilla no admite más de un carácter y que además no estén entre 1 y 9.

Más precisamente, verifica si la tecla pulsada es un digito dentro del rango especificado, y en caso de que no lo sea, ignora el evento del teclado.

```
(cuadrilla[a][b].getText().length() >= 1)
```

Sobre esa línea de código del método anterior, limitamos al usuario a colocar un solo digito.

Por otro lado tenemos el método `comprobarVictoria()` que comprueba si todas las columnas y filas de la cuadrilla son correctos. Este método está llamando a los métodos `comprobarSumaFila(int fila)` y `comprobarSumaColumna(int columna)` :

```
private boolean comprobarVictoria() {

        boolean ret = true;
        for (int i = 0; i < constanteEsquina; i++) {
for(int j=0;j<constanteEsquina;j++) {
                  ret=ret&&(comprobarSumaFila(i)&&comprobarSumaColumna(j));
            }
```

```
        }
return ret;      }
```

Estos métodos anteriormente nombrados, verifican que si la suma que realizo el usuario es correcta con respecto a la resolución del juego.

```
      private boolean comprobarSumaFila(int fila) {
            int suma = 0; boolean ret1= true;
            boolean ret2= true;

            for (int i = 0; i < constanteEsquina; i++) { int imput =
                  Integer.parseInt(cuadrilla[fila][i].getText() +
"0"); suma = Logica.sumaNumeros(suma, imput / 10);
                  ret2 = ret2 && !cuadrilla[fila][i].getText().isEmpty();
            } ret1 = ret1 && Logica.equalsNumeros(suma,
            Integer.parseInt(LabelSumaFila[fila].getText()));
            return ret1&&ret2;
      }
```

```
*El método comprobarSumaColumna(int columna) trabaja de forma similar a
comprobarSumaFila(int fila), solo que trabaja con las columnas de la cuadrilla
(ver proyecto).
```

Como podemos observar, estos métodos también verifican, tanto para las filas como para las columnas, que los textFields de las cuadrillas no sean vacíos.

```
ret2 = ret2 && !cuadrilla[fila][i].getText().isEmpty(); *
```

Sobre esa línea de código podemos observar que, mediante un acumulador booleano inicializado en “true” comprobamos si algún input de la cuadrilla está vacío, y en caso de que lo este, por mecánica de dicho acumulador booleano, devolverá false. De esta forma obligamos al usuario a completar todos los inputs de la cuadrilla para poder ganar el juego.

Tanto como para los labels que están a la derecha de la cuadrilla, como para los que están abajo, realizamos métodos para que se pinten de color verde en caso de que las sumas de los números ingresados por el usuario sean correctas:

```
      private void pintarFila(int fila,Color f) {
```

```
        LabelSumaFila[fila].setForeground(f);;
    }

    private void pintarColu(int columna, Color c) {
        LabelSumaColu[columna].setForeground(c);
    }
```

En caso de que el usuario se rinda y se quiere verificar que el juego era posible de resolver, sobre los inputs se colocara una posible resolución generada sobre la cuadrilla:

```
    private void rendirseYmostrarResolucion() { if (time.isRunning()) { for
        (int i = 0; i < constanteEsquina; i++) { for (int j = 0; j <
        constanteEsquina; j++) {
        cuadrilla[i][j].setText(cuadrilla[i][j].getName());
        cuadrilla[i][j].setBackground(colorRosa);
                    }
                    time.stop();
              }
        }
```

Por otra parte, además implementamos un sistema de record, que en el cual, dependiendo la dificultad que elija el usuario (fácil, normal, difícil), guarda el mejor tiempo correspondiente, en archivos separados, en formato .txt

En cuanto al botonReiniciar(), su función principal es la de darle al jugador, la posibilidad de volver a jugar, o iniciar un juego nuevo sin la necesidad de tener que estar cerrando el mismo y volverlo a abrir. Algunos detalles que tuvimos que contemplar, para este método es el hecho de vaciar las variables, que pueden contener información de una partida anterior, como lo es en el caso de, los hashMap que contienen la suma de las filas y columnas que se muestran en los laterales, y las banderas de dificultad que se utilizan para poder chequear en que dificultad esta parado el jugador. Además trabaja con la visibilidad de los elementos (labels y botones), ocultando y mostrando las correspondiente, dando la ilusión al usuario de que cambia de escenas o de pantallas.