Programación III - COMISIÓN 02 - Universidad de General Sarmiento

- Trabajo Práctico 1 -

1er. Cuatrimestre 2023

Profesores

Daniel Bertraccini - Gabriel Carrillo

Alumnos

Cristian Pereyra López - Lautaro Lombardi - Jeremías Ale Carbajales

INTRODUCCIÓN

En el presente trabajo práctico planteamos una posible solución para el análisis e implementación del Juego Aritmético dado como consigna del trabajo práctico nro 1. En el mismo intentamos utilizar todos los conceptos y herramientas aprendidos en clase con fines utilitarios y pragmáticos para la mejor resolución de todos los problemas que fuimos encontrando en su desarrollo. Procuramos utilizar y sacar provecho en todo momento de los pilares de la POO que vimos en materias anteriores como encapsulamiento y abstracción, y tuvimos en cuenta a la hora de la implementación por sobre todo el concepto de *"separated presentation"*, es decir, el código que se utiliza para la interfaz debe estar bien encapsulado y separado del código de negocio que ejecuta las funciones esenciales del juego. Nos ajustaremos al patrón de diseño MVC separando en GUI (Graphical User Interface) lo referido a la parte visual, Logic lo pertinente a lógica de negocio y Persistence a lo correspondiente a persistencia de datos.

La concepción de nuestro diseño está basada en lograr la mayor cohesión entre los métodos de cada clase buscando que estén fuertemente relacionados entre sí, delegando la responsabilidad de cada proceso a cual capa y clase según corresponda, y el menor acoplamiento posible, o sea el menor conocimiento entre las capas posible para poder funcionar y teniendo responsabilidades bien definidas, siendo más fáciles de entender por separado, reutilizables y adaptables a diferentes usos.

También procuramos crear un código limpio y entendible, evitando el código repetido mediante el uso de estructuras de control de flujo como ciclos for o for each, operadores ternarios o creando métodos para acciones recurrentes y utilizando convenciones de buenas prácticas al nombrar clases y variables teniendo como referencia los textos leídos en la cursada al hacer nuestras designaciones.

PRESENTACIÓN - REGLAS DEL JUEGO

Se presenta al usuario una grilla de n × n (6 >= n >= 4) según la dificultad elegida, con números enteros asociados a las filas y las columnas. El jugador debe resolver qué número poner en cada casillero de la grilla, de modo tal que la suma de los números de cada fila sea igual al valor dado en el extremo derecho, y que la suma de los números de cada columna sea igual al valor dado en el extremo inferior.

Por ejemplo, la Figura 1 muestra a la izquierda una grilla que se puede presentar al usuario al inicio del juego, y a la derecha el juego resuelto exitosamente.

Figura 1: Instancia de ejemplo, y juego resuelto.

Para resolver exitosamente el juego se debe completar la grilla logrando que todas las filas y columnas sumen su valor indicado respectivamente. El usuario no puede ingresar el 0 ni un número mayor a dos dígitos. En nuestra implementación se debe tener en cuenta que una misma grilla puede tener más de una solución posible (ya que la generación de valores designados como resultado de la suma se dan a consecuencia de un cálculo que involucra una aleatoriedad en los valores base) y la misma debe estar completa en su totalidad para que se valide la correcta resolución de la matriz entera.

IMPLEMENTACIÓN

A partir de nuestra elección como patrón de diseño del modelado MVC y su idea de *“separated presentation”* se optó por realizar una división del código en tres paquetes: por un lado tenemos un paquete de GUI con la interfaz de usuario que contiene la clases pertinentes a las distintas ventanas del juego, otro paquete de la lógica con las clases relativas a la lógica de negocio en donde se encuentra una clase que crea los valores y elementos para el funcionamiento del core del juego, y por último un paquete de Persistencia en donde se ubica la clase que interviene en el guardado y recuperación de datos de los jugadores.

Visual /IGU

Decidimos crear tres ventanas; ***StartCreen***: el inicio en donde el jugador ingresa su nombre (aquí se chequea que el input sea válido) y selecciona uno de los tres niveles en un JComboBox. Finalmente, un botón para empezar el juego que antes de pasar a la próxima ventana muestra un JDialog en el cual se explican las reglas del juego. ***GameScreen:*** la ventana principal del juego que tendrá un tamaño acorde al de la matriz de acuerdo al nivel elegido, un cronómetro en la parte superior, la matriz con casilleros a completar por el usuario y los paneles de chequeo de filas y columnas los cuales serán verdes o rojos de acuerdo a si la suma de los elementos respectivos son correctos o no. ***FinalScreen:*** la ventana de final de juego tiene un label dinámico como encabezado que dirá si el jugador ganó o perdió, la visualización (también dinámica) de los cinco mejores puntajes del juego junto al nombre de cada jugador y el nivel en el que lo hizo y dos botones al pie para volver a jugar o salir.

Lógica de Negocio/Logic

En cuanto a lógica se decidió implementar el juego a través de una matriz de valores. Al iniciar el juego, se genera una matriz de muestra completa a la cual en principio asignamos en cada posición números aleatorios, los cuales son inmediatamente sumados en filas y columnas para calcular los números resultado a partir de los cuales el jugador debe decidir los números a llenar en los casilleros al jugar. En base a esa matriz se genera una segunda matriz vacía, la cual recibirá los valores que el usuario pase durante el transcurso del juego y dos arreglos booleanos que se irán actualizando en base a qué columnas/filas se vayan completando y que están correspondidos con otros dos arreglos que guardan los resultados de las sumas de la matriz modelo.

A cada segundo que pasa el juego chequea en su lógica a la matriz vacía a ver si el usuario realiza algún cambio, y actualiza la matriz interna con los valores agregados/quitados. Si alguna fila/columna es completada y la suma de dichos valores da el resultado correspondiente, se guarda en el arreglo booleano en la posición correspondiente a su

respectivo resultado. Toda esta información, la guarda y se la envía a la interfaz, para que haga los cambios necesarios a nivel visual.

También en este paquete se encuentra la clase Jugador el cual es creado utilizando datos ingresados en la visual y se deriva a la Persistencia para ser guardado y eventualmente recuperado.

Persistencia/Persistence

En este paquete ubicamos lo pertinente al almacenamiento de los jugadores con sus nombres, puntaje y nivel. La lógica interna de esta capa se ocupa del almacenamiento individual de cada jugador que termina el juego, la recuperación de la totalidad de los jugadores y sus datos que jugaron previamente, el ordenamiento en base a sus puntajes y la creación de una lista con los cinco mejores para permitir la visualización en la pantalla final.

La interacción entre las tres capas es la adecuado al fin del funcionamiento de la aplicación, y acotado a lo respectivo de cada una; por ejemplo el ingreso en el input de usuario es validado en la IGU, una vez transcurrido el juego principal es enviado a la Lógica de Negocio para la creación de una instancia del objeto Player junto a su puntaje y nivel, luego derivado a la Persistencia para su almacenamiento. Finalmente en la última ventana del juego, la IGU pide a la Lógica de Negocios la lista de mejores puntajes, ésta interactúa con la Persistencia para que haga el proceso de recuperar la lista de jugadores del archivo. txt, ordenarlos y devolver una lista a la GUI para poder mostrar al usuario el top.

PROBLEMAS ENCONTRADOS

Algunos de los problemas encontrados, más allá de los relativos al proceso al detalle de implementación y codificación, algunos ejemplos son destacados son estos:

- **Validación del input en el ingreso de nombre de jugador**

  Investigando y probando código llegamos a aprender como utilizar un método auxiliar que capture tanto los caracteres ingresados y el texto completo del JText para validar que no se ingresen más de diez caracteres y que el nombre no sea de una longitud menor a tres ni que sea vacío.

```
textNombre.addKeyListener(new KeyAdapter() {
            @Override
            public void keyTyped(KeyEvent e) {
                if (textNombre.getText().trim().length() == 10) {
                    e.consume();
                }
            }
        });
```

```
private boolean validateName(String text) {
                  if (text.length() < 3 || text == null ||
                      text.trim().isEmpty()) {
                    return false;
                }
                return true;
            }
```

- **Validación del input en el ingreso los valores propuestos en la matriz de juego principal:**

  Utilizamos una método auxiliar para chequear que el usuario sólo ingrese números pero que no pueda ingresar “0”, pero si “10”, “20”, etc.

```
private void entryValidation(JTextField jText) {
        jText.addKeyListener(new KeyAdapter() {
            @Override
            public void keyTyped(KeyEvent e) {
                int key = e.getKeyChar();
                boolean numeros = key >= 48 && key <= 57;
                  if ((!numeros || jText.getText().trim().length() ==
2) || (key == 48 && jText.getText().trim().length() == 0)){
                    e.consume();
                }
            }
        });
    }
```

- **Creación de la Matriz Visual:**

  En un principio nuestra implementación tenia una creación hardcodeada de la matriz, eventualmente con la necesidad de crear distintos tamaños de matriz de acuerdo a los niveles, codificamos una resolución dinámica mediante estructuras control de flujo, lo cual no solamente nos clarificó el código sino también al hacerlo de forma dinámica pudimos usar solo una ventana para mostrarle al usuario la matriz del tamaño correspondiente. Dicha implementación la replicamos en la creación de otros elementos lo cual nos quitó código repetido.

```
JTextFieldsMat = new JTextField[matSize][matSize];
int JTextHorizontalPosition = 100;
int JTextVerticalPosition = 70;
for (int i = 0; i < matSize; i++) {
    for (int j = 0; j < matSize; j++) {
        JTextField textField = new JTextField();
        entryValidation(textField);
        textField.setHorizontalAlignment(SwingConstants.CENTER);
        textField.setFont(new Font("Tahoma", Font.BOLD, 25));
                  textField.setBounds(JTextHorizontalPosition,
             JTextVerticalPosition, 50, 43);
        contentPane.add(textField);
        textField.setColumns(10);
        JTextFieldsMat[i][j] = textField;
        JTextHorizontalPosition = JTextHorizontalPosition + 100;
    }
    JTextVerticalPosition = JTextVerticalPosition + 100;
    JTextHorizontalPosition = 100;
}
```

- **Guardado y recupero de datos:**

  Este es un tema que tuvimos que aprender de cero, particularmente los problemas con lo que nos encontramos además de buscar información sobre esto fue como registrar y recuperar los datos de cada jugador y luego ordenar de acuerdo a su puntaje. Ideamos guardar los jugadores en líneas y separar cada atributo mediante una coma, para luego poder al ir leyendo línea a línea y mediante el método .*split* recuperar los datos de los jugadores, ordenarlos y crear una nueva lista con los mejores puntajes a mostrar el la pantalla final de la GUI.

```
    while ((line = buffer.readLine()) != null) {
                    String[] parts = line.split(",");
                    String name = parts[0];
                    int score = Integer.parseInt(parts[1]);
                    String level = parts[2];

                    Player player = new Player();
                    player.setName(name);
                    player.setLevel(level);
                    player.setScore(score);
                    players.add(player);
                }
```

```
    Collections.sort(players, new Comparator<Player>() {
                @Override
                public int compare(Player player1, Player player2) {
                    return player1.getScore() - player1.getScore();
                }
            });
```

- **Uso de Switch / Operador Ternario:**

  En la implementación nos dimos cuenta que muchas veces nos quedaban porciones de código muy largas cuando utilizábamos condicionales con varios if/else. Lo resolvimos utilizando el mecanismo de control de selección Switch u Operador ternario según el caso y logramos un código más reducido.

```
            if (colSum == columnsResults[col]) {
                completedColumns[col] = true;
            } else {
                completedColumns[col] = false;
            }
completedColumns[col] = (colSum == columnsResults[col]) ? true : false;
```

- **Idioma y nombramientos:**

  Durante el trabajo colaborativo encontrábamos diferencias entre la forma de nombrar los elementos que hacía cada uno, así que para tener una versión final coherente y homogénea decidimos utilizar el inglés como estándar en común y pactar entre los tres en la nomenclatura de variables, clases, etc.

LINK A REPOSITORIO DE GITHUB

https://github.com/cristianpl99/TP-1-Juegos-Aritmeticos