Programación 3

Informe

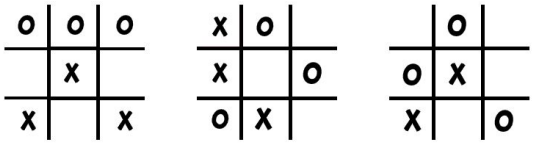
Gabriela Fernández - Javier Santillán - Nicolás Luna

Introducción

El objetivo del proyecto es crear una aplicación de tateti toroidal para dos jugadores totalmente funcional aplicada a una interfaz gráfica. La aplicación debe proporcionar un mecanismo adecuado para que los jugadores especifiquen sus jugadas por turnos. Además, debe detectar cuando alguno de los jugadores ha ganado e informar en la interfaz.

¿Que es un tateti toroidal?

Según como se explica en el enunciado, un tateti toroidal consta de un tablero 3X3 en que se cumplen las condiciones normales de victoria (horizontal, vertical, y diagonal) pero con un agregado y es que la diagonal puede continuar desde el otro extremo del tablero, como por ejemplo:



Separación de capas y denominación de clases

Nuestra aplicación es capaz de:

- Pedir nombre y fotos a un par de jugadores.
- Se puede elegir las dimensiones del tateti 3x3, 4x4, 5x5.
- Se asigna la marca de cada jugador de forma aleatoria.
- Se elige quien comienza de forma aleatoria.
- Cuando se llega a una victoria o un empate (posible en *4x4* y *5x5*) se muestran al ganador, su marca, y la cantidad de jugadas totales.
- Se puede reiniciar el tablero, iniciar otra partida o salir en medio de una partida usando.

Para la implementación de la aplicación se decidió separar en dos capas:

- Lógica *(que más adelante se la referencia como capa de negocio)*
- GUI *(que más adelante se la referencia como capa de visual)*

Estas partes son responsables de un aspecto de la aplicación, la capa funciones se encarga de la parte lógica como ganador(), la capa gui se encarga de todo lo visual que interactúa directamente con el jugador.

Capa Lógica:

- **Jugador:** Contiene el nombre de un jugador y su marca, si no se ingresa un nombre se autocompleta (jugador 1 o 2).
- **Tablero:** Esta clase modela un tablero con dimensiones cuadradas que se pueden especificar.
- **Tateti:** Contiene un tablero y dos jugadores, elige aleatoriamente la marca. también se encarga de ingresar jugadas al tablero y verificar si alguien gana o empata.

Capa GUI:

- **Inicio:** Inicia la aplicación.
- **v_presentacion:** El inicio del tateti, solo contiene un botón de inicio.
- **v_data:** Pide el nombre a los jugadores y da la opción de cambiar las dimensiones del tablero.
- **v_juego:** Muestra los jugadores, de quién es el turno, y genera un tablero de botones para jugar.
- **v_fin:** Muestra quién ganó y da información de los jugadores. También da la opción de jugar de nuevo, jugar otra partida con nuevos jugadores o salir.
- **Créditos:** Créditos de los participantes.
- **Acerca_De:** Da info del juego.

Desarrollo y decisiones de implementación

En la capa de negocio:

Se optó por implementar en primer lugar la **clase Tablero**, la cual representa solamente un tablero de tateti con una funcionalidad básica, este es representado mediante una matriz cuadrada de tipos enteros (int) la cual va a poder tener 3 estados (0 = espacio libre, 1 = espacio ocupado por el primer jugador, 2 = espacio ocupado por el segundo jugador).

Contiene la siguiente interfaz con 5 operaciones básicas:

public void ingresar(entero jugador, entero fila, entero columna) // Alta

public void quitar(entero fila, entero columna) // Baja

public int obtener (entero fila, entero columna) // Consulta

public void limpiarTablero() //Limpia el tablero

public int getDimensiones() // Devuelve la dimensión

Por otro lado tenemos la **clase Jugador** la cual representa a un Jugador junto con sus atributos (nombre, marca) y tiene la siguiente interfaz:

public getNombre() // Devuelve el nombre del jugador

public getMarca() // Devuelve la marca del jugador

public SetNombre() // Setea el nombre pasado por parámetro, si no se le pasa

ningún nombre, setea con un nombre genérico

public SetMarca() // Setea la marca con una marca aleatoria

Por último tenemos la **clase Tateti,** esta posee la mayoría de la lógica del juego en sí y fue donde tuvimos los principales dificultades a la hora de implementar la misma tiene la siguiente interfaz:

public int getTurno() // Devuelve el turno actual

private int setTurno() // Genera un turno inicial random

public boolean verificarTurno(entero jugador) // Verifica que el turno actual sea el correcto

public void cambiarTurno() // Cambia el turno al realizarse una jugada

public boolean jugar(entero jugador, entero fila, entero columna) // Devuelve true si la jugada se realizó correctamente si no devuelve false.

public boolean ganador(Integer jugador) // devuelve true si el jugador gano si no devuelve false

public Jugador getJugador1() // Devuelve al jugador 1

public Jugador getJugador2() // Devuelve al jugador 2

public String getJugadorActual() // Devuelve el nombre del jugador actual

public String getMarcaActual() // Devuelve la marca del jugador actual

public int getJugadas() // Devuelve la cantidad total de jugadas

public boolean ganoHorizontal (int jugador) // Devuelve true si ganó de forma horizontal

public boolean ganoVertical (int jugador) // Devuelve true si ganó de forma vertical

public boolean ganoToroidal (int jugador) // Devuelve true si gano diagonal o toroidal

A continuación se nombraran los métodos que tienen más importancia y se dará una breve explicación de los mismos.

public boolean ganoHorizontal(int jugador) y public boolean ganoVertical(int jugador):

La idea es que si se repiten las mismas marcas en toda una columna o fila completa entonces el jugador habrá ganado. Para ello el algoritmo que implementamos fue pararnos en un principio sobre:

- En ganoVertical(): la primera columna a forma de pivote y movernos sobre las filas, comparando el número de jugador obtenido de la fila y la columna pivote con el número de jugador pasado por parámetro.
- En ganoHorizontal(): la primera fila como un pivote y luego ir moviéndonos en las columnas comparando el número de jugador que se encuentra en cada columna de la fila pivote con el número de jugador pasado por parámetro.

Si son iguales se incrementará un contador. Si al finalizar de recorrerla el contador es igual a la dimensión del tablero entonces se retorna true, si no, se cambiará a la columna o fila pivote siguiente y volver a repetir el mismo algoritmo, si al finalizar de recorrer todas las columnas o filas no hubo ganador devolverá false.

public boolean ganoToroidal(int Jugador):

Este método cubre los casos en diagonal y diagonal toroidal, para eso se recorre la primera fila y al encontrar una *marca del jugador* procede a hacer una llamada a un método recursivo que “bloquea” la columna en la que se encontraba la anterior marca para no volver a recorrer sobre esa columna, procede a hacer llamadas recursivas e implementa un OR aumentando la *fila* y también aumenta o disminuye la *columna* dependiendo la situación, el caso base del método recursivo se da cuando se llega a la última fila.

En la capa visual:

Como ya lo hemos mencionado la capa gui se encarga de la parte visual, por lo tanto es la responsable de crear las distintas ventanas que va a tener el juego.

La principal dificultad que encontramos a la hora de hacer la parte gráfica de la aplicación fue cómo poder implementar un tablero de botones que represente el tablero del tateti sin la necesidad de repetir tanto código. Para ello nuestra solución fue implementar una matriz cuadrada de botones.

Desarrollo del algoritmo toroidal

En un principio ganoToroidal() implementó un algoritmo que buscaba que un jugador tuviera al menos una marca en cada fila y columna del tablero.

```
public boolean ganoToroidal(Integer jugador) {
        int contador=0;
        boolean primercondicion = true;
        int columna=0; // columna == Y
        while(columna < 3) {
                int filaAux=0;
                contador=0;
                while (filaAux < 3 && contador < 1) {
                        if (tablero[filaAux][columna]==jugador) {
                                contador++;
                        }
                        filaAux++;
                }
                primercondicion = primercondicion && (contador > 0 ? true : false);
                columna++;
        }

        boolean segundacondicion = true;
        int fila=0; // fila == X
        while(fila < 3 && primercondicion) { //necesita que se cumpla la primera condicion
                int columnaAux=0;
                contador=0;
                while (columnaAux < 3 && contador < 1) {
                        if (tablero[fila][columnaAux]==jugador) {
                                contador++;
                        }
                        columnaAux++;
                }
                segundacondicion = segundacondicion && (contador > 0 ? true : false);
                fila++;
        }
        return primercondicion && segundacondicion;
}
```

Implementado cubría los casos de victoria pero también varios casos que no cumplian, en resumen falsos positivos.

En un siguiente versión de este método se consiguió cubrir todos los casos, en la misma se usaba un array booleano que se usaba para bloquear las columnas ya ocupadas para que no las vuelva a recorrer aunque en realidad si lo hacía solo que si encontraba que estaba bloqueada la ignoraba y seguía con el ciclo.

```
public boolean ganoToroidal(int jugador) {
        if (jugador!=1 && jugador!=2)
                throw new IllegalArgumentException("el jugador no existe");
        return ganoToroidal(jugador, 0, new boolean[getDimensiones()]);
}

public boolean ganoToroidal(int jugador, int fila, boolean[] columnasOcupadas) {
        boolean gano = false;
        int columna = 0;

        if (fila==getDimensiones()-1) {
                while (columna<getDimensiones()) {
                        if (tablero.obtener(fila, columna)==jugador && !columnasOcupadas[columna]) {
                                gano=true;
                        }
                        columna++;
                }
        }
        if (fila<getDimensiones()-1){
                while(columna<getDimensiones()) {
                        if (tablero.obtener(fila, columna)==jugador && !columnasOcupadas[columna]) {
                                columnasOcupadas[columna]=true;
                                if(ganoToroidal(jugador, fila+1, columnasOcupadas)) {
                                        gano = true;
                                }else {
                                        columnasOcupadas[columna]=false;
                                }
                        }
                        columna++;
                }
        }
        return gano;
}
```

Luego de agregar la posibilidad de crear un tablero que pudiera tener varias dimensiones surgió otro problema, el método no se acoplaban a un 4x4 o 5x5, falsos positivos otra vez.

Luego de cambios en casi todo el código, lo único que se conservó fue la idea de usar un array de booleanos para “bloquear” las columnas ya ocupadas se consiguió dar con un algoritmo que satisficiera los casos toroidales en un 3x3 tanto como en 4x4 o 5x5 (y suponemos que para matrices más grandes también)

```
public boolean ganoToroidal(int jugador) {
	int columna=0;
	boolean acumB=false;
	while (columna<getDimensiones()) {
		acumB = acumB || ganoToroidal(jugador, 0, columna, new boolean[getDimensiones()]);
		columna++;
	}
	return acumB;
}


private boolean ganoToroidal(int jugador, int fila, int columna, boolean[] bloqueadas) {
	if(fila == getDimensiones()-1) {
		return tablero.obtener(fila, columna) == jugador;
	}
	if (fila < getDimensiones()-1) {
		if (columna < getDimensiones()-1 && tablero.obtener(fila, columna) != jugador) {
			return false;
		}else {
			if (tablero.obtener(fila, columna) == jugador && bloqueadas[columna]==false) {
				bloqueadas[columna]=true;
				if (columna == getDimensiones()-1) {
					return ( ganoToroidal(jugador, fila+1, 0, bloqueadas)
						|| ganoToroidal(jugador, fila+1, columna-1, bloqueadas) );
				}else {
					if (columna == 0) {
						return ( ganoToroidal(jugador, fila+1, getDimensiones()-1, bloqueadas)
							|| ganoToroidal(jugador, fila+1, columna+1, bloqueadas) );
					}else {
						return ( ganoToroidal(jugador, fila+1, columna+1, bloqueadas)
							|| ganoToroidal(jugador, fila+1, columna-1, bloqueadas) );
					}
				}
			}
		}
	}
	return false;
}
```

En este último método se quitaron los ciclos while y se recorre de la mejor forma la matriz disminuyendo las iteraciones innecesarias.

ganoToroidal(int jugador): recorre la primera fila en busca del jugador cuando lo encuentra llama al método recursivo, en caso contrario retorna falso.

ganoToroidal(int jugador, int fila, int columna, boolean[] bloqueadas):
Caso base: Si se posiciona en la última fila verifica que en la fila y columna que recibió se encuentre al jugador.
Caso recursivo: La lógica básica es que mientras encuentre encuentre al jugador y la columna no esté bloqueada se vaya aumentando fila en cada llamada recursiva cuando se da el caso que está en alguna columna que no sea un extremo (osea columna igual a 0 o a la longitud del tablero) se hacen llamadas recursivas a los laterales de la columna en la que se encuentra parado, en los casos de extremos como estando en la columna = 0 se hace el llamado aumentando columna y revisando en el extremo opuesto.

Siempre que se hace el llamado recursivo antes se actualiza el array de booleanos pasando a true la columna en la que se encontró al jugador en esa iteración.
Cuando se da la situación que no se encuentra al jugador en toda una fila se retorna false.

Conclusión

Pudimos comprobar durante el desarrollo del proyecto que al separar la lógica de lo visual y elegir nombres descriptivos para sus clases, funciones o métodos, así como también elegir qué tarea específica iban a realizar en cada clase, se nos hacia mas fácil ver que parte del juego fallaba e ir directo a las funciones que encontrábamos defectuosas para luego modificarlas.

La implementación de los tests unitarios nos ayudaron rápidamente a identificar si los métodos de la capa de negocio cumplían bien su funcionamiento y a actualizar el código cuando teníamos que cambiarlo.