Universidad Nacional de General Sarmiento

Programación III
Trabajo Práctico N° 1

Integrantes:

- Menegol, Tomás.
- Mendoza, Leonardo.
- Cristian Brizuela.
- Sebastian Vivas.

Cursada:

- 2do Semestre 2025.

Comisión:

- 1, Turno noche.

Fecha de entrega:

- Martes 15/09/2025.

Introducción:

El presente documento tiene como objetivo plasmar en escrito los procesos, desarrollos y decisiones del primer trabajo práctico de la materia Programación 3, correspondiente al segundo cuatrimestre.

El proyecto consta de la implementación de un juego “Nonograma”. El mismo se juega sobre una cuadrícula en blanco y negro. Cada casilla de la cuadrícula puede estar pintada de negro, marcada con una X roja, o dejada en blanco (color default). Las pistas numéricas, ubicadas a la izquierda de cada fila y en la parte superior de cada columna, indican la cantidad de casillas que deben ser marcadas de negro por cada fila y columna para resolver el juego correctamente.

Herramientas:

El trabajo se desarrolló en lenguaje Java versión 17, utilizando Swing y la herramienta WindowBuilder, con el IDE Eclipse.

El codigo se organiza bajo la arquitectura “forms and controls” enseñada en clase, donde se busca principalmente respetar el principio de “separated presentation”. Parte de este principio implica que en ningún momento el código de negocio llame al código de la interfaz. Esto incluye que el código de negocio no conozca ni la identidad o las tecnologías utilizadas por la interfaz.

El proyecto consta de tres paquetes, entre ellos: un paquete para la interfaz, otro para la lógica de negocio y un tercero para el main que dispara el juego. Procedemos a listar el funcionamiento de los mismos y las clases correspondientes.

Problemáticas encontradas:

El primer desafío a enfrentar fue la división del trabajo. En este sentido, decidimos partir lo solicitado por el trabajo en problemas más pequeños y afrontarlos por partes. Nos dividimos en 2 equipos, uno encargado de la lógica, y otro de la interfaz, decidiendo que como probablemente el equipo de la interfaz finalice primero con su trabajo, todos nos reuniriamos para continuar con lo que quede de la lógica de negocio.

Una vez se decidió esto, cada equipo armó los requerimientos necesarios para representar su parte del problema (una especie de mini TAD). Dado que todavía no utilizamos TDD, previo a escribir las primeras líneas de código, decidimos probar algunos de los algoritmos a implementar (en especial los más complejos) en papel, para posterior poder directamente copiarlos sobre las clases requeridas. Esto sucedió por ejemplo con la función "acumularCasillasNegrasPorFila".

También se aprovecharon las clases de consulta para resolver dudas más propias a la implementación del juego. Por ejemplo, a fines de la lógica de negocio, para que un juego este resuelto de manera satisfactoria, lo unico que importa es que la casilla este marcada de color negro. Entonces decidimos que esto es lo único que íbamos a corregir en dicha instancia (esto NO quiere decir que la lógica de negocio no va a contemplar los otros 2 estados de la grilla: Blanco y Marcado).

Otra duda relacionada se originó en cuanto a cómo representar los 3 posibles estados de cada casilla. Primero se discute utilizar la herramienta Enum, pero posterior se decidió utilizar un sistema más simple con ints, donde:

- 0: Casilla blanca.
- 1: Casilla negra.
- 2: Casilla marcada.

Está decisión se tomó dado a que dejar las celdas negras con el valor "1", nos ayudaba para sumar rápidamente cuántas celdas negras consecutivas hay en la grilla (siendo estas una suma de 1 por cada celda consecutiva).

La problemática final y más grande ronda en cuanto a cómo corregir un nonograma resuelto por el usuario. Esto se da debido a que un nonograma podría tener más de una respuesta correcta, en especial al escalar el tamaño de la grilla. Después de unos días de deliberación, llegamos a la siguiente solución. El paso a paso del juego va a ser el siguiente:

1. Se generará una matriz cuadrada, vacía.
2. Se rellenará la grilla con celdas negras a través de algún algoritmo. Esto finalizara nuestra grilla "resuelta".
3. Otro algoritmo se encargará de recorrer la grilla resuelta por fila y por columna, contando las celdas negras, para generar las referencias que utilizara el usuario para resolver el juego.
4. Se generará otra matriz vacía, que será en la que el usuario irá ingresando el estado de cada casilla.

Dado que de todos modos tenemos que armar un algoritmo que me genera referencias para una grilla con celdas negras, podemos correr nuevamente este algoritmo sobre la grilla que completara el usuario. De está manera, si las referencias de ambas grillas coinciden, no importa cual sea el contenido en sí de cada grilla (porque pueden haber múltiples soluciones correctas), sabemos que la generada por el usuario es una solución válida.

Paquete Lógica:

Este paquete contendrá toda la lógica de negocio. Empezamos por declarar una clase abstracta "Tabla". Está tendrá los métodos comunes a todos los tableros, es decir, getters, setters y el método encargado de generar las referencias en base a una grilla. Adicional, una tabla bajo nuestra definición está conformada por una grilla y sus referencias (referencias para las columnas y para las filas) con lo cual tendrá estas 3 variables de instancia.

La clase "TableroRespuesta" será una subclase que se generará con celdas negras de manera aleatoria, para posteriormente generar referencias y que el usuario pueda utilizarlas para jugar.

Un factor que creemos que vale la pena resaltar es el approach al momento de generar un juego de nonograma de forma pseudo aleatoria. Tras varios juegos previo y durante el desarrollo, notamos como en la mayoría aproximadamente el 50% de las casillas totales eran negras, esto nos llevó a idear un método, ya que vamos a dar con un número previamente calculado de casillas negras totales la aleatorización era una cuestión de límites claros y distribución.

Por un lado se genera un patrón aleatorio en el rango de las filas, de tal forma que al “rellenar” recorriendo por filas, no se recorran en un orden secuencial, aumentando la aleatoriedad pero también mitigando la aglomeración de casillas negras en ciertos puntos del tablero. Además se estableció una referencia a la cantidad de casillas negras disponibles, y se establece una regla, si la cantidad de casillas **disponibles** es mayor que el 50%, aproximado, de la **cantidad total** de casillas negras, entonces el maximo y minimo de casillas negras que pueden ponerse por fila será mayor. Del mismo modo si nos encontramos por debajo del 50% estos serán menores. Dentro de cada fila se rellena de forma aleatoria solo las casillas negras asignadas para esta fila particular, y se le resta a la referencia de casillas negras **disponibles** aquellas que han sido utilizadas.

Este método nos permite un tablero con casillas negras dispersas, con cruces seguros por las altas cantidades de casillas negras en las primeras iteraciones (posibilitando un juego más justo y con menos situaciones de azar), con un máximo siempre de **n-1**, siendo **n** el largo de la fila, y la facilidad de no tener que implementar un rellenado por columnas.

Por otro lado la clase "TableroUsuario" será otra subclase, con una grilla vacía, la cual el usuario irá rellenando casilla por casilla en el juego. Al final del mismo, en base al estado de la grilla se generarán las referencias, las cuáles se corregirán contra las de TableroRespuesta.

Por último la clase "Juego" será la encargada de tomar las solicitudes brindadas por una interfaz y dictar el ritmo del juego, es decir, la generación de las

grillas, la comparación que indica si el juego fue resuelto satisfactoriamente, etcétera.

Paquete Interfaz:

El paquete posee las clases responsables de las visualizaciones del usuario. Se comienza por la clase “Frame” la cual, como su nombre lo indica, extiende a la clase “JFrame” y nos permitirá definir el size de la ventana del juego, mostrar las distintas vistas del juego y sobre todo interactuar con la lógica. La clase “Frame" recibe la instancia de “Juego” correspondiente a la partida actual. Al instanciar una nueva vista del juego cada una recibe el mismo “Frame”, no solo para compartir el mismo juego si no para poder generar las acciones requeridas según los inputs del usuario.

Posteriormente tenemos a las antes mencionadas “vistas” o “clases de vista”, si así se les puede llamar. Estas clases de vista extienden a “JPanel” y representan cada una de las instancias del juego. La primera es “vistaMenuInicio” la cual presenta el inicio de la GUI, mostrando un título, la opción para conseguir instrucciones de como jugar, el botón que da inicio al juego y las opciones de “tamaño” del tablero (5x5, 10x10 y 15x15). Una característica de esta vista es que reserva en una variable el tamaño elegido por el usuario para poder llamar al método de frame “generarJuego()” el cual genera los tableros internos de la lógica con el tamaño deseado.

Cada una de estas vistas es generada desde Frame y se maneja un diccionario de vistas, para que se pueda generar el cambio entre ellas. Para ello se valida si la vista actual no es la vista a la que se quiere cambiar, y de ser el caso se borran todos los elementos de la vista y se instancia la nueva vista.

Por otro lado la clase “vistaJugar” muestra el juego en sí, con el tablero del usuario y las referencias correspondientes así como otras funcionalidades. Dentro de la misma clase se pudo configurar los elementos que representan el tablero y sus referencias para poder escalar de tamaño automáticamente, dependiendo del tamaño seleccionado para jugar. Así mismo, tal como sus nombres lo indican, “vistaComoJugar” despliega un texto que explica las reglas y origen del nonograma y “vistaGanar” junto con “vistaPerder” se muestran tras un resultado positivo o negativo, respectivamente, al terminar el juego. Es en estas últimas donde se pudo implementar un **extra**, tratándose de mostrar la solución correcta al perder el juego.