Reporte Segundo Proyecto Programacion

MATCOM

2022 – 2023

Implementacion de un juego de Domino

Adrian Hernandez Santos – C211
Alejandro Alvarez Lamazares – C212

Sobre el juego:

Nuestra implementación del juego de dominó está basada en el juego clásico, tanto doble 9, como doble 7, para la implementación inicial, las reglas, etc. Partiendo de ahí pudimos determinar distintos conceptos base para modificar y crear un juego más extensible que su variante clásica. Apoyándonos en los conceptos básicos de Programación Orientada a Objetos aprendidos en el curso de Programación, obtuvimos las herramientas necesarias para crear un proyecto base en el que cualquiera en un futuro puede llegar e implementar nuevas funcionalidades con un mínimo de código en la mayoria de los casos, siempre que esté definida una abstracción para dicho elemento a extender.

Toma de decisiones:

Como se puede apreciar en el historial del repositorio en Github, hubo una versión inicial del proyecto, en la que se aplicó un conjunto de decisiones erróneas con respecto a la **POO** y **SOLID**. Específicamente fue el hecho de aplicar dichos patrones al extremo, como por ejemplo, pensar la forma óptima de crear abstracciones desde la base: por ejemplo, creando una abstracción básica para un juego, en la cual cualquier cosa que admitiera una ficha, un tablero y un conjunto de reglas fuera dicho objeto. El problema con esto es que se llego a tener una cantidad absurda de clases e interfaces, que a medida que se seguian abstrayendo dichos conceptos, se complejizaba su interoperabilidad e incluso el papel de algunas dentro de la lógica de la aplicación. Dicho sea de paso, lógica que existía solamente en concepto, pues la aplicación no se ejecutaba, por el simple hecho de que como fue dicho anteriormente, al haber tantas piezas y aplicar mal los patrones de **SOLID**, se abstraían los conceptos fundamentales (y los no tan esenciales) hasta tal punto de nunca llegar a conformar una versión primaria que funcionara, ni siquiera parcialmente.

Luego de esta odisea, se llegó a la conclusión de cambiar de paradigma y centrarse en la solucion al problema, aunque no fuera la óptima estructuralmente. Adoptando un diseño básico con el minimo de clases necesarias y múltiples responsabilidades para cada una, pero teniendo clara la interacción inicial entre cada componente del programa, se logró crear la primera versión funcional y jugable del juego de dominó. Basándonos en las clases **DominoToken**, **DominoPlayer** y **DominoGame** inicialmente, de las cuales fueron extendidas las relativas a las fichas y a los jugadores, pues nos dimos cuenta que el juego podia contener la lógica que ejecutaria dicho juego, valga la redundancia. Así pudimos complementar todas las piezas del puzzle y a partir de ahi, ir fragmentando la estructura del codigo y la jerarquia, para que fuera extensible y aplicando el patron **DRY**, para evitar errores duplicados de codigo semejante.

Aspectos clave:

- Fichas
- Jugadores
- Condición de Ganado
- Ganador
- Juego
- Interfaz gráfica

Fichas:

Las fichas en el dominó son la parte más importante, puesto que en ellas se basa todo el sistema de puntaje, los posibles movimientos y demas funcionalidades que permiten el correcto funcionamiento del juego. Para el nuestro, decidimos implementar 3 variantes:

DominoToken:

Representa la ficha estándar del juego de dominó. Su cálculo de valor esta dado por la suma de los valores de cada componente (izquierda y derecha) como en el juego normal. Además de poseer un sistema para generar las fichas dada una numeración de juego (doble 9 por ejemplo), el cual es heredado a cada una de las fichas hijas para que tenga la misma funcionalidad, dependiendo del tipo de ficha en cuestión.

SixUnvaluableDominoToken:

Representa una ficha en la cual el valor 6 está penalizado. Esto quiere decir que si en alguna de sus componentes aparece un 6, el valor del mismo automáticamente pasa a ser 0, o sea, no tiene valor.

DoubledValueDominoToken:

Representa una ficha en la cual el valor de cada una de sus componentes es el doble de lo que tuviera si fuera una implementación estándar de dicha ficha. Por ejemplo, el doble 3, al ser sus 2 componentes 3 y 3, se convierte (representativamente) en un doble 6, puesto que el valor de cada una se duplica, pero sigue siendo un 3 a la hora de jugarla.

Jugadores:

El jugador es una parte clave en el juego de dominó. Sin él, el propio concepto de juego no tendría sentido. La funcionalidad básica del jugador seria jugar una ficha en su turno, dado un estado del juego, dependiendo de su implementación. Dicha funcionalidad tiene 2 momentos clave, por una parte la ficha inicial, si el jugador es el que comienza la partida, y por otro, jugar una ficha en su turno, despues de una cierta cantidad de jugadas. Por su importancia en el propio juego fue que decidimos hacer varias implementaciones de estos, en específico 4:

RandomDominoPlayer:

Un jugador que seleccionara una ficha aleatoria para jugar de las posibles jugadas que pueda hacer, basado en el sistema **RNG** de **.NET**.

GreedyDominoPlayer:

Un jugador que jugará siempre la ficha con mayor valor de entre las posibles jugadas que tenga en mano, independientemente de la forma de calcular el valor del tipo de ficha que se utilice en el juego.

HumanDominoPlayer:

Implementación de un jugador humano, se le pedirá al usuario que dado un estado del juego y una mano de posibles fichas, realice una jugada en su turno, con una entrada en la terminal de la consola del juego.

DataDominoPlayer:

Jugador que basará su jugada primeramente seleccionando de una data (número con mayor ocurrencia entre sus fichas), y posteriormente, en caso de no poder jugar por su data, jugará la primera ficha que sea posible.

Condición de Ganado:

La condición de ganado en el dominó determina cuando un juego dado debe terminar. Independientemente del juego que se juegue, luego de cada jugada de un jugador, debe verificarse el estado de la partida, para decidir si se debe continuar o se debe terminar dicho juego. Hemos realizado 2 implementaciones para esta funcionalidad:

StandardWinCondition:

Condición de ganado estándar, o sea, las reglas básicas para que una partida termine: que nadie pueda hacer una jugada o que algún jugador se quede sin fichas en la mano.

FourRoundsWinCondition:

Condición de ganado que consiste en que el juego finaliza luego de 4 rondas, o cuando un jugador se queda sin fichas. Una ronda está definida como la sucesión de jugadas desde un jugador a sí mismo nuevamente, o sea, un ciclo completo de jugadas.

Ganador:

Como es de esperar, en un juego de varios jugadores, debe haber al menos un ganador (digo al menos porque hasta ahora no hemos tenido en cuenta el concepto de empate, pero podría existir). Dicho ganador, dependiendo de las reglas, puede estar determinado por diferentes factores, siempre luego de que el juego finaliza (ahí entra la condición de ganado). Hemos hecho 2 implementaciones de cómo seleccionar un ganador:

MinValueWinner:

Esta implementación selecciona al jugador con menor valor entre todas sus fichas para declararlo ganador. Dicho valor viene calculado con la implementación de ficha que se esté utilizando en la partida en cuestión.

MoreTokensWinner:

Esta implementación selecciona al ganador como el jugador con mayor cantidad de fichas luego de que se acaba la partida, sin tomar en cuenta el valor de las mismas, solo la cantidad en mano.

Juego:

DominoGame:

Esta clase es el centro de control de toda la lógica del juego. En ella se concentra la interacción entre todos los componentes de la aplicación, que luego de que se le pasan una serie de argumentos predefinidos por el usuario, utiliza la genericidad brindada por la abstracción para separarse de el tipo final que desea utilizar el usuario. Dicha clase se encarga de repartir las fichas a los jugadores, de llevar un estado global que comparte con los elementos de control (condicion de ganado y selección del ganador) y mediante el método principal **Result** se encarga de hacer funcionar toda la maquinaria implementada en las clases previamente explicadas. A pesar de haber deseado una mayor modularidad, no encontramos una mejor forma de juntar todos los elementos, puesto que usar algo como un árbitro, además de agregar una capa extra de complejidad en el código, obligaba a separar el estado del juego de la propia clase **DominoGame**, por lo que tendríamos que haber adoptado otra forma de atacar el problema.

Interfaz gráfica:

La interfaz gráfica de nuestro juego esta compuesta por un script bastante simple que ejecuta una aplicación de consola. Dicha aplicación está compuesta por un conjunto de funciones que se encargan de, dada una entrada válida del usuario, crear una configuracion previa al juego para luego instanciar este y proceder a dejar que el usuario pueda jugar su partida.

Los métodos encargados de hacer esa “pre-configuración” del estado inicial de la partida son los siguientes:

SetPlayer Count
SetValueAmount
SetTokensPerPlayer
SetPlayers
SetWinCondition
SetWinner
SetToken

Cada uno configura el aspecto del juego especificado en su nombre, retornando un entero, un array de **DominoPlayer**, o una instancia de una clase que implementa una interfaz,

dependiendo el caso, que posteriormente son usados como estado inicial para la partida a ejecutarse.

PrepareGame:

Este método instancia un nuevo juego, dados los parametros previamente entrados por el usuario.

GetInput:

Este método auxiliar fue creado con el objetivo de utilizar el patrón **DRY** (Don’t Repeat Yourself), puesto que en cada método anterior se pedía una entrada del usuario explícitamente. Además, se le agregó un parámetro opcional que permite acotar el resultado obtenido a valores con sentido y capturar ademas, entradas no numéricas, en este caso números enteros.

Main:

El método **Main** de dicho script, simplemente se encarga de mostrar una información básica explicando los parámetros a modificar y ejecutar los métodos previamente comentados, para crear un flujo correcto de interaccion entre el usuario y la aplicación final.