Diseño de Software Orientado a Objetos

Trabajo Práctico Obligatorio 2

Integrantes:
- Lara Combina
- Ignacio Dias Gundin

Captura de Requisitos

- El juego de la vida consiste en un “juego” sin jugadores, donde la única entrada es el estado inicial de las células en un tablero.
- El tablero consiste en una matriz de N x M celdas, donde en cada casilla existe una célula con dos posibles estados: viva o muerta.
- A partir de un conjunto de reglas aplicadas sobre las células se determina el próximo estado del tablero. Las reglas son las siguientes:
    - Si una célula está viva y tiene dos o tres vecinos a su alrededor se mantiene viva.
    - Si una célula está muerta y tiene menos de tres o más de tres vecinos a su alrededor se mantiene muerta.
    - Si una célula está viva y tiene menos de dos vecinos, o más de tres, se muere.
    - Si una célula está muerta y tiene tres vecinos, esta pasa a estar viva.

-El tablero generado deberá ser mostrado en consola.

Patrones Utilizados

Observer (Salida en consola)

En esta primera versión nuestro juego tiene una única forma de mostrar la salida, que es a través de la pantalla. Creemos que esto a futuro podría cambiar, pidiéndonos que mostremos una salida mediante una GUI, o que además de la salida se nos requiera recopilar estadísticas del juego, por eso es que decidimos aplicar el patrón Observer de manera temprana.
Este patrón utilizará a Game como clase observada (“subject”) y la interfaz IOutput como observadora (“observer”). La clase ConsoleDisplay implementará IOuput y una interfaz IObserver, siendo el motivo por el cual tiene los métodos show() y update().

Command (Reglas del Juego)

A cada célula se le aplica un conjunto de reglas para modificarlas, las cuales además podrían verse modificadas a futuro. Para implementar esto, la idea consiste en que el tablero tenga como atributo un conjunto de reglas, que pueden cambiarse dinámicamente.

Clases

- **Cell** (implementa **ICell**). Las celulas tienen un atributo CellType type. Este **CellType** es un enumerado con los valores DEAD y ALIVE.
- **Board** (implementa **IBoard**). La posición de las células se representa con una matriz estática de N x M. El tablero debe ser inicializado con un estado inicial, en nuestro caso donde todas las células tienen el mismo estado pasado como parámetro. Además, el tablero guarda todas las reglas a aplicar sobre las células, basada en la cantidad de vecinos que tiene. Los métodos disponibles del tablero serán los siguientes:
    - putCell(int row, int col):void
    - getCell(int row, int col):ICell
    - nextState():IBoard (retorna el proximo tablero en donde se ha aplicado las reglas a todas las células).
    - numberOfNeighbours: int
- **Game**. Esta clase será observada por la consola. Los métodos públicos son:
    - Initialize(board)
    - play(int numberOfGenerations): void (se ejecuta n veces la regla de avanzar el tablero)
    - notifyObservers(): este método se comunicará con las posibles salidas, informando el estado del tablero actual.
- **ConsoleOutput** (implementa **GameObserver**): mostrará por pantalla los resultados.
- I**Rule.** Las reglas abarcan todas las posibilidades de estados o transformaciones, por lo que a toda célula se le debe aplicar exactamente una regla. En nuestro caso las clases de reglas son **RuleStayAlive, RuleStayDead, RuleBorn** (cambia de muerta a viva)**,** y **RuleDie** (cambia de viva a muerta).

Segunda Parte

Nuevos Requerimientos

- Implementar las versiones "Colorised life" del game of life, en donde las células tienen distintos colores. La implementación debe soportar las variantes Immigration y Quadlife.
- Implementar las versiones denominadas Generations. Su implementación debe soportar las versiones Brian's Brain y Star Wars.
- Permitir al usuario cambiar la velocidad de avance del juego.
- Implementar un modo simulación, en el que se va avanzando un paso a la vez. El usuario debe presionar una tecla para avanzar al siguiente paso.
- Implemente un modo simulación acotada, en el que se avance hasta un número K de pasos, dado por el usuario.
- Permitir guardar la salida en archivos. Guardar cada paso de una simulación acotada en un archivo diferente.

Las versiones ‘Colorised Life’ introducen colores en las células, pero no modifican su comportamiento. Las células en la variante Immigration, por ejemplo, tienen dos colores, rojo y azul, y las reglas son iguales a las del juego original, con la diferencia de que cada vez que nace una nueva célula esta toma el color de la mayoría de sus vecinos (cómo puede nacer con tres células, toma el color de las dos células con color repetido a su alrededor o la de todos sus vecinos). Por otro lado está la variante Quadlife, en donde las células pueden tomar cuatro colores (rojo, amarillo, verde, azul). Comparte las mismas reglas que Immigration, pero cuando nace una nueva célula, si no hay un color que sea mayoría (es decir, que los tres vecinos sean de colores distintos), entonces la célula que nace es del color restante.

Las versiones ‘Generations’ introducen distintas etapas de vida en las células, por lo que pueden ‘envejecer’. En la variante ‘Brian’s Brain’ las células nacen cuando tienen dos células vivas alrededor. En la siguiente generación, independientemente de sus vecinos, pasan a un segundo estado ‘moribundas’, donde no se las considera vivas. Una generación después, pueden revivir o morir completamente. Alternativamente, en la variante ‘Star Wars’, las células tienen cuatro estados posibles. Una célula en estado 0 se convertirá en estado 1 si y sólo si tiene dos vecinos en estado 1. Una célula en estado 1 no cambiará si tiene 3, 4 o 5 vecinos en estado 1; de lo contrario, entrará en estado 2 en el siguiente tic y luego en estado 3 antes de morir.

Cambios Realizados

- Cambiamos la clase Game por un GameController. Esto se debe a que ahora necesitábamos tener distintos modos para controlar la velocidad y el número de generaciones creadas para un tablero. GameController continúa teniendo un método play, pero en su definición este es ejecutado por
- Introducimos la posibilidad de haber variantes al juego original. Por lo tanto, para permitirlas en nuestro programas tenemos la clase GameCreator, que genera una instancia del juego con la variante elegida en base a los parámetros que pasa un usuario mediante un archivo Properties.
- Por cada variante de juego tenemos un conjunto de clases de reglas. Luego con un RuleFactory se decide el conjunto de reglas que instancia el juego desde GameCreator.
- Lo último que cambia, pero ligeramente del diseño original, es que incrementamos el número de estados posibles que puede tener una célula a partir del enumerado CellType. Por otro lado, en el tablero tenemos una función más general para obtener los vecinos a una célula. Antes devolvíamos solamente el número de vecinos vivos, ahora retornamos un diccionario donde las claves son los tipos de células y el valor el número de ellas. Esto es necesario para hacer distintos tipos de comprobaciones en las reglas.

Nuevos Patrones de Diseños Agregados

Facade

Tendremos por un lado la clase main que solo tendrá un objeto Game, y será el encargado de comunicarse con todas las clases necesarias para poder crear una partida.
Cada juego tiene una modalidad (Classic, Brian's Brain, Star Wars, etc.), un Display (consola, archivo) y un PlayStyle(número de generaciones, generación infinita con una velocidad, generación infinita paso a paso).

Factory Method

Desde GameCreator se llama a la mayoría de fábricas para generar clases del juego. Estas son BoardFactory, PlaybackmodeFactory, RulesFactory y DisplayFactory. Todas ellas toman al menos una cadena obtenida desde el archivo Properties para saber las propiedades del objeto a crear. Adicionalmente, BoardFactory llama a CellFactory.

Intentamos en un momento tener un patrón AbstractFactory para crear cada variante, pero siendo que principalmente lo que cambiaba entre cada juego eran los estados de las células y las reglas, al final no fue necesario.

Strategy

La clase GameController se considera Strategy, en donde una clase Main instancia un objeto de este tipo y luego utiliza los métodos ofrecidos por el objeto.
Las reglas también se implementan con este patrón, ya que producen algoritmos fácilmente intercambiables.

Principios de Diseño Aplicados

- **Encapsular lo que cambia**: en nuestro programa las cosas que cambian entre una variante y otra son las reglas, los métodos de salida del juego, y las formas en las que este se reproduce. Por lo tanto, para implementarlas usamos una combinación entre los patrones FactoryMethod y Observer.
- **Programar respecto de interfaces**: tenemos las interfaces IBoard, ICell, GameDisplay, GameObserver, GameController y Rule.
- **Favorecer composición sobre herencia:** nuestro programa no tiene herencia directamente, sino que lo logramos todo mediante interfaces y composición de objetos (véase la implementación de GameController).
- **Principio de Responsabilidad única, y clases con alta cohesión:** separamos las clases creadoras (factories) de las implementaciones, y reducimos significativamente el número de métodos que llama cada una.
- **Clases deben estar abiertas para extensión, pero cerradas para modificación:** ahora mismo, si queremos introducir una nueva variante, bastaría con crear un nuevo conjunto de reglas para ella, modificar la fábrica de reglas, y agregar nuevos tipos de célula de ser necesario. Lo mismo si se quisiera agregar un nuevo observador o modo de reproducción.

- **Depender de abstracciones**, **no depender de clases concretas**: debido a que tenemos las interfaces definidas, no dependemos de las clases concretas para utilizarlas dentro de todo el programa.