Nodo4

1. Descripción General

La clase Nodo4<Tipo> es una plantilla de clase que implementa un nodo en una estructura de datos cuadridireccional. A diferencia de los nodos de listas tradicionales (que solo apuntan a un siguiente elemento), un Nodo4 mantiene cuatro punteros a nodos vecinos: arriba, abajo, izquierda y derecha. Esta estructura es fundamental para construir una malla o una cuadrícula en el juego, permitiendo la navegación en todas las direcciones. Al ser una plantilla, puede almacenar cualquier tipo de dato, lo que lo hace versátil para representar diferentes elementos del tablero, como puntos o celdas.

2. Funcionalidad Clave

El rol principal del Nodo4 es actuar como un contenedor de datos y como un punto de enlace para sus vecinos. Su funcionalidad se centra en:
Almacenamiento de datos: Contiene un valor genérico de tipo Tipo.
Gestión de enlaces: Permite establecer y obtener punteros a los nodos vecinos en las cuatro direcciones. Esto es crucial para la navegación y la construcción de la malla cuadridireccional.
Flexibilidad: Su naturaleza de plantilla (template) le permite ser reutilizado para diferentes propósitos, como el Nodo4<Punto> que forma la malla de puntos del tablero y el Nodo4<Celda> que forma la malla de celdas.

3. Mecanismos y Lógica

El funcionamiento de la clase se basa en la manipulación de punteros. 🧭
Constructor: Inicializa el nodo con un valor valor para su dato. Además, todos sus punteros a vecinos (arriba, abajo, izquierda, derecha) se inicializan a nullptr. Esto garantiza que el nodo no apunte a direcciones de memoria inválidas y está listo para ser enlazado en la estructura.

Métodos establecer...: Estos métodos (por ejemplo, establecerArriba()) permiten asignar direcciones de memoria a los punteros del nodo. La lógica del programa principal (como la clase Tablero) es la encargada de llamar a estos métodos para construir la malla, asegurando que los enlaces sean correctos y bidireccionales cuando sea necesario (por ejemplo, nodoA->establecerAbajo(nodoB) y nodoB->establecerArriba(nodoA)).

Métodos obtener...: Estos métodos de acceso (por ejemplo, obtenerDerecha()) devuelven las direcciones de los punteros a los nodos vecinos. La lógica del programa principal los utiliza para navegar por la malla, permitiendo, por ejemplo, recorrer una fila de puntos de izquierda a derecha o mover una celda hacia abajo.

Métodos obtenerDato(): Se ha sobrecargado para permitir tanto la lectura (const Tipo&) como la modificación (Tipo&) del dato almacenado, ofreciendo un control flexible sobre el contenido del nodo.

4. Elementos de Programación

La clase Nodo4 utiliza los siguientes elementos de C++:

Plantillas (template): La clase está definida como template <typename Tipo>. Esto permite que Nodo4 se adapte para manejar cualquier tipo de dato sin la necesidad de reescribir el código, lo que fomenta la reutilización de código.

Punteros (*): La lógica central de la clase se basa en el uso de punteros para gestionar las referencias a los nodos vecinos.

nullptr: Se utiliza para inicializar los punteros a un valor seguro, indicando que no apuntan a ningún objeto válido.

Constructor: Un constructor parametrizado que inicializa los miembros de la clase al momento de la creación del objeto.

Encapsulamiento: Aunque los atributos son privados, se exponen a través de métodos públicos (getters y setters), controlando el acceso a los datos internos y a los punteros.

Sobrecarga de Métodos: El método obtenerDato() está sobrecargado para ofrecer versiones const y no-const, permitiendo que el compilador elija la versión adecuada según si el objeto es modificable o no.

Nodo Normal

1. Descripción General

La clase Nodo<T> es una plantilla de clase que representa un nodo en una estructura de datos lineal doblemente enlazada. Su principal función es almacenar un puntero a un dato de tipo genérico T y mantener enlaces a los nodos inmediatamente anterior y siguiente en la secuencia. Este diseño permite el recorrido en ambas direcciones, lo que es crucial para estructuras como pilas, colas y listas que necesitan insertar o eliminar elementos tanto al inicio como al final.

2. Funcionalidad Clave

El rol de Nodo<T> es servir como un bloque fundamental para construir estructuras de datos más complejas. Sus responsabilidades clave son:
Contenedor de datos: Almacena una referencia (T*) al dato que representa.

Gestión de enlaces: Mantiene punteros a los nodos adyacentes, lo que permite la navegación bidireccional a través de la lista.

Encapsulamiento: Sus miembros privados aseguran que solo las clases autorizadas (como Lista<T>) puedan manipular directamente sus enlaces, protegiendo así la integridad de la estructura de datos.

3. Mecanismos y Lógica

El funcionamiento de Nodo<T> se basa en la manipulación de punteros para establecer las conexiones:

Constructor: El constructor inicializa el nodo con un puntero a un dato (T* d) y establece los punteros siguiente y anterior a nullptr. Esto garantiza que un nodo recién creado está aislado y listo para ser enlazado en una lista.

Métodos get y set: Los métodos get (getSiguiente, getAnterior) permiten acceder a los nodos vecinos para realizar operaciones de recorrido. Los métodos set (setSiguiente, setAnterior) son utilizados por la clase Lista<T> para construir o modificar los enlaces de la estructura. Al ser friend, la clase Lista<T> tiene acceso directo a los miembros privados de Nodo<T>, lo que facilita una manipulación eficiente y segura.

4. Elementos de Programación
La clase Nodo<T> utiliza los siguientes elementos de C++:
Plantillas (template): La clase está definida como template <typename T>, lo que la convierte en una clase genérica. Esto permite que el mismo código de Nodo pueda ser utilizado para almacenar punteros a Jugador, PowerUp o cualquier otro tipo de dato, promoviendo la reutilización del código.
Punteros (*): La lógica central de la clase se basa en el uso de punteros para gestionar las referencias a los datos y a los nodos vecinos.

nullptr: Se utiliza para inicializar los punteros a un valor seguro, indicando que no hay un nodo anterior o siguiente en un extremo de la lista.

friend: La declaración template <typename> friend class Lista; otorga a la clase Lista acceso a los miembros privados de Nodo, lo que es esencial para que Lista pueda construir y modificar correctamente los enlaces de sus nodos internos.

Constructor de inicialización: El constructor utiliza una lista de inicialización (: dato(d), siguiente(nullptr), anterior(nullptr)) para inicializar directamente los miembros de la clase, lo que es una práctica recomendada en C++ para mejorar el rendimiento y la legibilidad.

Lista Enlazada
1. Descripción General
La clase ListaEnlazada<Tipo> es una plantilla de clase que implementa una lista lineal genérica. A diferencia de un arreglo estático, una lista enlazada puede crecer o decrecer dinámicamente. Esta implementación utiliza los nodos cuadridireccionales (Nodo4), lo que la hace ideal para las estructuras de listas usadas en el tablero. La clase se encarga de gestionar la adición de nodos y el acceso a los datos, proporcionando una capa de abstracción sobre la compleja manipulación de punteros de los nodos individuales.

2. Funcionalidad Clave
ListaEnlazada<Tipo> es la responsable de manejar colecciones de elementos de manera eficiente. Sus funciones principales son:

Gestión de colecciones: Permite almacenar y organizar nodos de forma secuencial.

Inserción dinámica: Añade nuevos elementos al inicio o al final de la lista, sin la necesidad de un tamaño predefinido.

Acceso a la lista: Proporciona acceso al primer nodo de la lista (cabeza) y a la cantidad total de elementos.

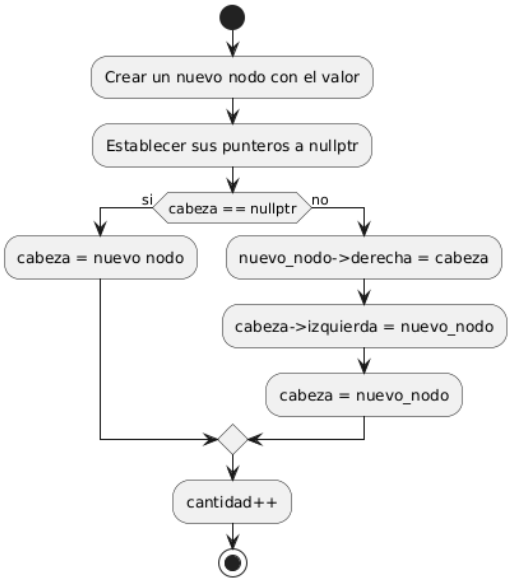
Gestión de memoria: Se encarga de la creación y destrucción de los nodos, evitando fugas de memoria.

3. Mecanismos y Lógica
La lógica de esta clase se basa en el manejo del puntero a la cabeza (cabeza) y en la iteración a través de la lista.

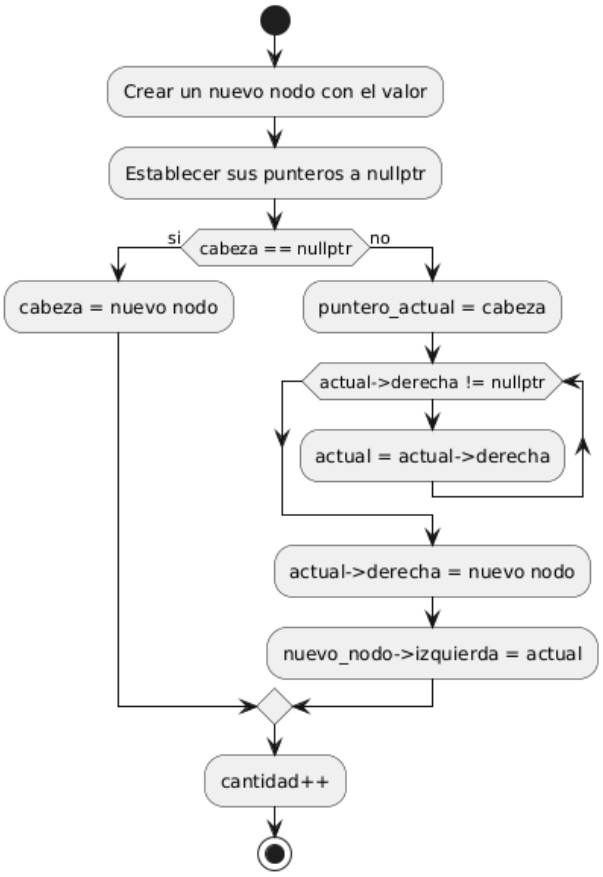
Constructor y Destructor: El constructor inicializa el puntero cabeza a nullptr y la cantidad a 0. El destructor invoca el método limpiar() para liberar toda la memoria de los nodos, una práctica crucial para evitar fugas de memoria.

Inserción (insertarInicio y insertarFinal):

insertarInicio() crea un nuevo nodo y lo coloca al frente. Si la lista no está vacía, enlaza el nuevo nodo con el anterior cabeza y viceversa, asegurando una conexión bidireccional.



insertarFinal() es más compleja. Utiliza un bucle while para recorrer la lista desde cabeza hasta encontrar el último nodo (aquel cuyo puntero derecha es nullptr). Una vez que lo encuentra, establece los enlaces para el nuevo nodo.



Recorrido: La navegación por la lista se realiza a través de un puntero temporal (actual) que se mueve de nodo en nodo utilizando el método obtenerDerecha() de Nodo4.

limpiar(): Este método itera sobre todos los nodos, liberando la memoria de cada uno con delete antes de avanzar al siguiente. Finalmente, reinicia la lista estableciendo cabeza a nullptr y cantidad a 0.

4. Elementos de Programación
Esta clase utiliza los siguientes elementos de C++:

Plantillas (template): La clase es una plantilla (template <typename Tipo>), lo que la convierte en una clase genérica. Esto permite que el mismo código se use para listas de Jugador, PowerUp, Linea, o cualquier otro tipo de dato.

Punteros (*): La clase depende de punteros para gestionar las conexiones entre los nodos. La lógica de inserción y recorrido se basa en la manipulación de estos punteros.

nullptr: Se utiliza para verificar si la lista está vacía (cabeza == nullptr) o si se ha llegado al final de ella.

Constructores y Destructores: El constructor asegura un estado inicial válido, mientras que el destructor y el método limpiar() manejan la gestión de memoria de manera manual, una tarea esencial en C++ para evitar problemas de memoria.

Clases relacionadas: La clase ListaEnlazada tiene una relación de dependencia y composición con la clase Nodo4, ya que crea y gestiona instancias de ella.

Cola

1. Descripción General

La clase Cola<T> es una plantilla de clase que implementa una estructura de datos de tipo FIFO (First-In, First-Out - Primero en entrar, primero en salir). Su diseño se basa en una lista doblemente enlazada, lo que le permite manejar eficientemente operaciones de inserción y eliminación tanto en la parte frontal como en la trasera. Esta clase es esencial para gestionar el flujo de turnos de los jugadores en el juego, garantizando que cada jugador tenga su turno de manera equitativa.

2. Funcionalidad Clave

El rol de Cola<T> en la aplicación es crucial para la gestión del estado del juego. Su funcionalidad se centra en:

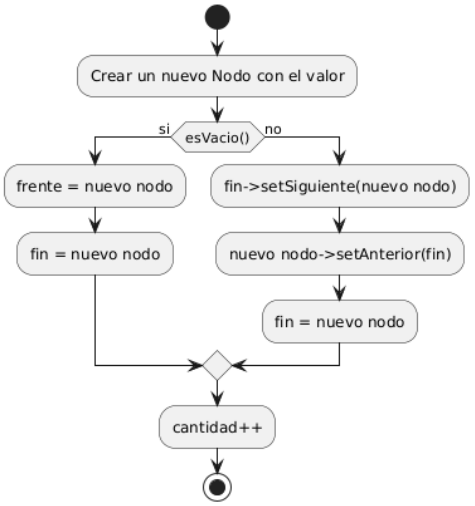
Gestión de turnos: Organiza los turnos de los jugadores, permitiendo que cada uno tenga su oportunidad de jugar en el orden correcto.

Manejo de estados especiales: El método encolarFrente es una funcionalidad específica para el poder especial de Doble Línea, permitiendo a un jugador volver a jugar de inmediato.
Acceso y verificación: Permite ver el elemento al frente de la cola y verificar si está vacía.
Gestión de memoria: La clase es responsable de la creación y destrucción de los nodos, previniendo fugas de memoria.

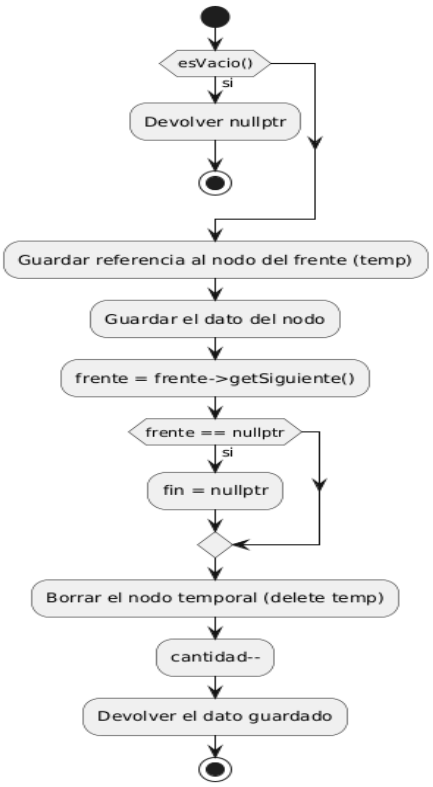
3. Mecanismos y Lógica

La lógica de la clase Cola se basa en la manipulación de dos punteros clave: frente y fin, que apuntan al primer y último nodo de la lista, respectivamente.

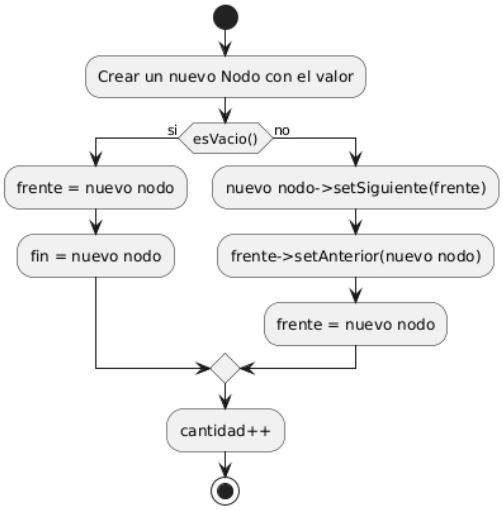
Constructor y Destructor: El constructor inicializa los punteros frente y fin a nullptr y la cantidad a 0. El destructor garantiza que todos los nodos se liberen de la memoria, llamando al método desencolar repetidamente hasta que la cola esté vacía.
encolar(): Este método añade un nuevo nodo al final de la cola. Si la cola está vacía, el nuevo nodo se convierte en el frente y el fin. Si no, el nuevo nodo se enlaza al final actual y el puntero fin se actualiza.



desencolar(): Este método remueve y devuelve el nodo al frente de la cola. Al eliminarlo, actualiza el puntero frente al siguiente nodo. Si este era el último nodo, también se actualiza el puntero fin.



encolarFrente(): Esta función especial añade un nodo al frente de la cola, insertándolo antes del nodo actual en el frente. Esto es lo que permite que el jugador que usó el poder de Doble Línea tome su turno de nuevo.



4. Elementos de Programación
La clase Cola utiliza los siguientes elementos de C++:

Plantillas (template): Al igual que otras clases de estructuras de datos, Cola es una plantilla (template <typename T>) para garantizar su reutilización con cualquier tipo de dato, como Jugador en este caso.

Punteros y Nodos (*, Nodo): La clase utiliza punteros a Nodo<T> para construir la estructura de la cola, y los métodos setAnterior y setSiguiente de Nodo para gestionar los enlaces bidireccionales.

nullptr: Se utiliza para inicializar los punteros y para verificar si la cola está vacía o si se ha llegado al final.

Constructor y Destructor: El constructor asegura que la cola comience en un estado válido, mientras que el destructor maneja la liberación de la memoria de los nodos.

Uso de Clases de Apoyo: La clase Cola depende directamente de la clase Nodo, demostrando la composición de un sistema modular.

Pila
1. Descripción General
La clase Pila<T> es una plantilla de clase que implementa una estructura de datos de tipo LIFO (Last-In, First-Out - Último en entrar, primero en salir). Su diseño se basa en una lista doblemente enlazada, pero su lógica de operaciones de entrada y salida la restringe a un solo extremo: el tope. En el contexto de tu juego, la clase Pila se utiliza específicamente para gestionar los PowerUps que tiene un jugador, asegurando que el último PowerUp obtenido sea el primero en ser utilizado.

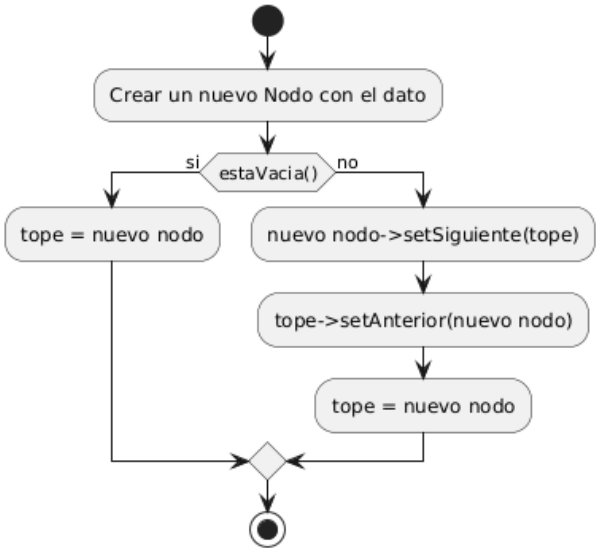
2. Funcionalidad Clave
La funcionalidad de la clase Pila se centra en las operaciones básicas de una pila y es crucial para el sistema de PowerUps. Sus funciones principales son:

Gestión de PowerUps: Almacena y administra los poderes especiales que un jugador puede obtener durante el juego.
Adición de elementos (apilar): Permite añadir nuevos PowerUps al tope de la pila.
Eliminación de elementos (desapilar): Remueve y devuelve el PowerUp que está en el tope de la pila.
Consulta del tope (verTope): Permite ver el PowerUp que está en la cima sin removerlo.
Verificación del estado: Proporciona un método para comprobar si la pila está vacía, lo que es esencial para saber si el jugador tiene poderes disponibles.

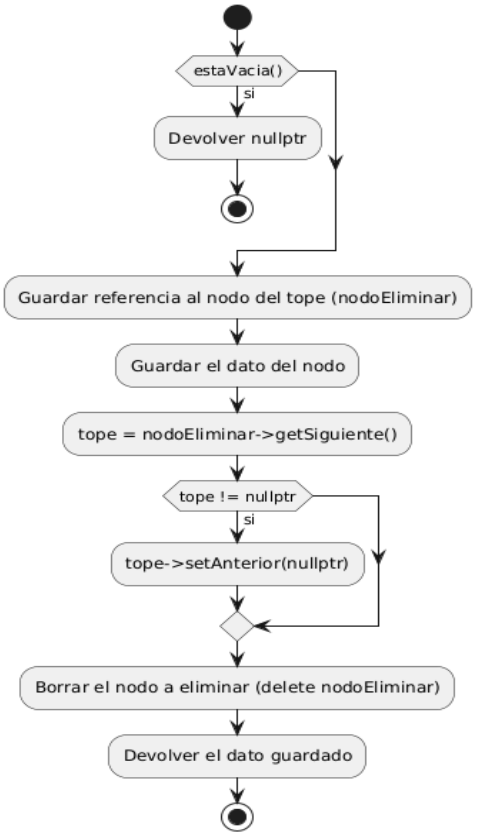
3. Mecanismos y Lógica
La lógica de Pila se basa en la manipulación de un único puntero principal: tope, que siempre apunta al primer nodo de la pila.

Constructor y Destructor: El constructor inicializa el puntero tope a nullptr, lo que significa que la pila está vacía al ser creada. El destructor garantiza una gestión de memoria segura al llamar a desapilar() repetidamente hasta que todos los nodos sean liberados.

apilar(): Este método crea un nuevo nodo y lo inserta en el frente de la lista enlazada, lo que se convierte en el nuevo tope. El nodo anterior se convierte en el siguiente del nuevo nodo.



desapilar(): Este método elimina el nodo en el frente de la lista (el tope). Guarda el dato del nodo para devolverlo, actualiza el puntero tope al siguiente nodo y luego libera la memoria del nodo eliminado. Si la pila está vacía, devuelve nullptr para evitar errores.



verTope(): Simplemente devuelve el dato del nodo al que apunta tope sin modificar la estructura de la pila.

estaVacia(): Un simple método booleano que comprueba si tope es nullptr.

4. Elementos de Programación
Esta clase utiliza los siguientes elementos de C++:

Plantillas (template): La clase es genérica (template <typename T>), lo que la hace flexible para gestionar diferentes tipos de datos, en este caso, punteros a objetos PowerUp.

Punteros y Nodos (*, Nodo): La clase depende de punteros y de la clase Nodo<T> para construir la estructura LIFO.

nullptr: Es usado para inicializar y verificar el estado de la pila, evitando accesos a memoria no válida.

Constructor de inicialización: El constructor utiliza una lista de inicialización para garantizar que los miembros de la clase se inicialicen correctamente.

Gestión de memoria: La clase es responsable de la creación (new) y la eliminación (delete) de los nodos que contiene, lo cual es fundamental en C++ para prevenir fugas de memoria.

Lógica LIFO: La implementación de los métodos apilar y desapilar está diseñada para cumplir estrictamente con el principio LIFO.

Punto

1. Descripción General

La clase Punto es un elemento fundamental que representa las intersecciones o nodos de la cuadrícula del tablero. Cada Punto tiene una posición única, definida por una fila y una columna. Su rol principal es servir como un ancla para las Líneas y las Celdas del tablero, ya que son los elementos que se conectan para formar la estructura del juego. Además, cuenta con una propiedad que le permite ser "destruido", lo que es clave para implementar la lógica de los poderes explosivos.

2. Funcionalidad Clave

Punto es una clase de datos simple pero vital. Sus responsabilidades clave son:
Identificación espacial: Almacena las coordenadas de fila y columna, lo que le da su ubicación exacta en el tablero.
Gestión de estado: Permite cambiar su estado a "destruido" para representar los efectos de un PowerUp explosivo.
Representación visual: Proporciona un método para obtener el símbolo que debe representarse en el tablero ('o' para un punto normal y 'X' para uno destruido), lo que es esencial para la interfaz del juego.

3. Mecanismos y Lógica

La lógica de Punto es sencilla y se basa en la manipulación de sus atributos. No contiene algoritmos complejos.

Constructor: El constructor inicializa el Punto con una fila y columna dadas. La propiedad destruido se inicializa por defecto en false, lo que significa que un punto recién creado está intacto.

Métodos get: Los métodos getFila() y getColumna() permiten que otras partes del código accedan a la ubicación del punto.

Método destruir(): Este método es el único que modifica el estado del punto. Simplemente cambia el valor de la variable booleana destruido a true.

Método simbolo(): Utiliza un operador ternario (condicion ? valor_si_verdadero : valor_si_falso) para devolver el carácter adecuado según el estado de la variable destruido.

4. Elementos de Programación

Esta clase utiliza los siguientes elementos de C++:

Clase y Atributos: Es una clase sencilla con miembros de datos (fila, columna, destruido) y métodos para acceder y modificar su estado.

Constructor de inicialización: El constructor utiliza una lista de inicialización (: fila(fila), columna(columna), destruido(false)) para asignar los valores iniciales a sus miembros, lo que es una práctica recomendada en C++.

const: Los métodos getFila(), getColumna(), estaDestruido() y simbolo() están marcados como const, lo que garantiza que no modificarán el estado interno del objeto.

Tipo de dato booleano (bool): El uso de un bool para la variable destruido es una forma eficiente de representar un estado de dos opciones.

Linea

1. Descripción General

La clase Linea representa el segmento de conexión entre dos puntos adyacentes en el tablero. Es un elemento fundamental del juego, ya que su colocación por parte de un jugador es el movimiento principal de la partida. Cada Linea está definida por sus dos puntos finales (p1 y p2), su orientación (HORIZONTAL o VERTICAL), y su estado actual: si ha sido colocada y, de ser así, qué jugador la colocó.

2. Funcionalidad Clave

El rol de la clase Linea es crucial para la jugabilidad. Sus funciones principales son:
Definición espacial: Delimita los bordes de las celdas y define el camino entre los puntos.
Gestión de estado: Mantiene el rastro de si la línea ha sido dibujada (colocada) y por quién (jugadorColoco).
Representación visual: Proporciona el símbolo correcto para ser mostrado en el tablero, lo que permite a la interfaz del juego visualizar las líneas correctamente.
Verificación de conexión: Permite comprobar si la línea conecta un par específico de puntos, lo cual es vital para la lógica del GestorLineas.

3. Mecanismos y Lógica

La lógica de Linea es bastante directa, centrada en la manipulación de sus atributos para reflejar el estado del juego.

Constructor: El constructor inicializa la línea con punteros a sus dos puntos (inicio, fin) y su orientacion. Por defecto, una nueva línea no está colocada y no tiene un jugador asociado (jugadorColoco = ' ').

Métodos colocar() y quitar(): Estos métodos cambian el estado de la línea. colocar() establece la propiedad colocada a true y asigna la inicial del jugador que realizó la acción. quitar() revierte este proceso, permitiendo la eliminación de líneas.

toString(): Este método es una lógica de representación simple. Utiliza un condicional para devolver un string apropiado: "---" o "|" si la línea está colocada, o un espacio en blanco si no lo está. Esto hace que la clase sea auto-suficiente para su visualización.

conecta(): Este método implementa la lógica de verificación de conexiones. Recibe dos punteros a Punto y devuelve true si la línea conecta a esos dos puntos en cualquier orden. Esta doble verificación es importante porque una línea puede ser recorrida en ambas direcciones.

4. Elementos de Programación
Esta clase utiliza los siguientes elementos de C++:

Punteros (*): La clase Linea mantiene punteros a los objetos Punto que conecta. Esto es una asociación que permite a la línea acceder a la información de los puntos sin poseerlos.

Enumeraciones (enum class): La enumeración Orientacion define un conjunto de valores discretos (HORIZONTAL, VERTICAL), lo que mejora la legibilidad del código y evita el uso de "números mágicos" para representar el estado de la línea.

#include y declaraciones forward: La declaración class Punto; en el archivo de cabecera es una declaración forward. Esto permite que la clase Linea conozca la existencia de Punto sin tener que incluir todo su archivo de definición, lo que reduce las dependencias y el tiempo de compilación.

const: El uso de la palabra clave const en los getters y en el método conecta() garantiza que estos métodos no modifiquen el estado interno del objeto.

Celda
1. Descripción General
La clase Celda es la unidad fundamental de captura en el tablero. Representa un cuadrado en la cuadrícula del juego y es el objetivo final de los jugadores: completar sus cuatro lados para "capturarla". Cada Celda está definida por sus cuatro esquinas (Puntos) y sus cuatro líneas (Lineas). Su estado principal, completada, determina si ha sido capturada por un jugador, y si es así, se asocia con un objeto Jugador como su propietario.

2. Funcionalidad Clave
Celda es una clase vital que encapsula la lógica central del juego. Sus funciones principales son:
Estructuración del tablero: Funciona como un "nodo" que conecta cuatro puntos y cuatro líneas, estructurando la malla del tablero.
Manejo de estado: Rastrea si ha sido completada y por qué jugador, lo cual es la base del sistema de puntos.
Lógica de captura: Contiene el método verificarCompletada para comprobar si la celda puede ser capturada, y capturar para ejecutar la acción y asignar la propiedad.
Soporte para PowerUps: Puede contener un objeto PowerUp y proporciona métodos para asignarlo y recuperarlo.

3. Mecanismos y Lógica
La lógica de Celda se basa en la interacción con los objetos de otras clases a través de punteros. No crea estos objetos, sino que los utiliza.

Constructor: El constructor de la clase recibe punteros a los cuatro Puntos y las cuatro Lineas que la componen. Esta es una inyección de dependencias, lo que significa que la Celda no se encarga de crear sus componentes, sino que estos le son proporcionados por la clase Tablero. Inicializa su estado a completada = false y propietario = nullptr.

verificarCompletada(): Este método implementa la lógica de la victoria. Itera sobre el array lineas y llama al método estaColocada() de cada Linea. Si todas las líneas están colocadas, devuelve true, indicando que la celda está lista para ser capturada.

capturar(): Si la celda aún no ha sido completada, este método le asigna un propietario, establece completada a true y, lo más importante, llama a sumarPunto() en el objeto Jugador para actualizar la puntuación.

contieneLinea(): Es un método de utilidad que recorre su array de líneas para verificar si una Linea específica pertenece a la Celda, lo cual es útil para la lógica del GestorLineas.

obtenerInicialPropietario(): Devuelve la inicial del Jugador propietario si existe, o un espacio en blanco si no. Esta es una lógica de representación que facilita la impresión del tablero.

4. Elementos de Programación
Esta clase utiliza los siguientes elementos de C++:

Punteros (*): La clase Celda mantiene punteros a Puntos, Lineas, Jugador y PowerUp. Esto es una asociación con las otras clases, lo que le permite interactuar con ellas sin ser responsable de su ciclo de vida.

Arreglos de punteros: Los arreglos esquinas y lineas son arreglos de punteros, una forma eficiente de almacenar múltiples referencias a objetos relacionados.

const: El uso de const en los getters y en verificarCompletada() garantiza que estos métodos no alteren el estado de la Celda, adhiriéndose a las mejores prácticas de programación.

Constructores: Celda tiene un constructor por defecto y uno parametrizado que facilita la inicialización de los atributos.

Jugador
1. Descripción General
La clase Jugador es una de las entidades centrales del juego. Representa a cada participante y gestiona su estado, incluyendo su nombre, puntuación y los PowerUps que ha recolectado. Esta clase es crucial para rastrear el progreso de cada persona y para habilitar las acciones especiales que pueden realizar durante su turno.

2. Funcionalidad Clave
El rol de Jugador es encapsular toda la información y la lógica de un participante en la partida. Sus funciones principales son:
Identificación: Almacena el nombre y una inicial única para cada jugador.
Gestión de puntos: Rastrea la puntuación del jugador a medida que va capturando celdas.
Manejo de PowerUps: Administra la colección de poderes especiales del jugador usando una Pila, lo que asegura un comportamiento de último en entrar, primero en salir (LIFO).

Interacción: Proporciona los métodos necesarios para que otras clases del juego (como Celda y Juego) puedan interactuar con el jugador, por ejemplo, para sumarle puntos o pedirle que use un poder.

3. Mecanismos y Lógica
La lógica de Jugador se basa en la manipulación de sus atributos y en la interacción con otras estructuras de datos como Pila.

Constructor: El constructor inicializa al jugador con su nombre y color, y automáticamente calcula su inicial para una representación simplificada. La puntuación se inicia en 0.

calcularInicial(): Este es un método de utilidad privado que recorre el nombre del jugador para encontrar la primera letra alfabética y la convierte a mayúscula. Esto asegura que la inicial sea siempre un carácter válido y fácil de mostrar en el tablero.

sumarPunto() y restarPunto(): Estos métodos simples incrementan o decrementan la puntuación. La lógica de restarPunto incluye una validación para asegurarse de que los puntos nunca sean negativos.

Manejo de PowerUps:

agregarPowerUp(): Recibe un puntero a un PowerUp y lo apila en la pilaPowerUps del jugador.

usarPowerUp(): Llama a desapilar() en la pila para obtener el PowerUp del tope. Este es el comportamiento LIFO en acción.

tienePowerUps(): Una simple delegación a pilaPowerUps.estaVacia(), pero negando el resultado para dar una semántica más clara al lector.

mostrarPowerUps(): Este método es para la interfaz de usuario. Comprueba si el jugador tiene poderes y, si los tiene, llama al método imprimirPila() para mostrar los símbolos de los poderes disponibles.

4. Elementos de Programación
La clase Jugador utiliza los siguientes elementos de C++:

Pila (Pila<PowerUp>): Esta es la estructura de datos clave para gestionar los PowerUps. La relación de composición entre Jugador y Pila asegura que la pila se cree y se destruya con el jugador.

Punteros (*): La clase maneja punteros a PowerUps para agregar y usar poderes, permitiendo el polimorfismo cuando se aplican los efectos de los poderes.

string y char: Se utilizan para el nombre y la inicial, con funciones de la librería <cctype> como isalpha() y toupper() para el procesamiento de caracteres.

Constantes y const: MAX_POWERUPS y COLOR_DEFAULT son constantes para mejorar la legibilidad y la mantenibilidad del código. Los getters están marcados como const para garantizar que no modifiquen el estado del objeto.

Tablero
Metodos
~Tablero()
1. Descripción General
El destructor de la clase Tablero es una función crucial para la gestión de la memoria del juego. Su única y vital responsabilidad es liberar toda la memoria dinámica que fue asignada durante la vida del objeto Tablero. Esto incluye la malla de Puntos y las Lineas que se crearon para estructurar el tablero. Al realizar esta tarea de manera segura, el destructor previene las fugas de memoria, que son un problema común en la programación en C++.

2. Funcionalidad Clave
El destructor se encarga de dos tareas de limpieza principales:
Desasignar la malla de puntos: Elimina de la memoria cada Nodo4<Punto> que conforma la cuadrícula principal.
Desasignar las líneas: Borra cada Linea individual que se creó para los bordes del tablero.

3. Mecanismos y Lógica
La lógica del destructor es simple pero requiere un manejo cuidadoso de los punteros para evitar errores de memoria. Se basa en el recorrido completo de las dos estructuras de datos principales del tablero para liberar la memoria de cada nodo u objeto.
Liberación de la malla de puntos: El destructor utiliza un bucle anidado para recorrer la estructura cuadridireccional.
Un puntero de fila (filaPtr) se inicializa al inicio de la malla.
Un bucle externo (while (filaPtr)) avanza de fila en fila, moviendo filaPtr hacia abajo con filaPtr->obtenerAbajo().
Dentro de cada fila, un bucle interno (while (colPtr)) recorre los nodos de izquierda a derecha. Un puntero de columna (colPtr) se mueve de nodo a nodo con colPtr->obtenerDerecha().

Antes de avanzar, se usa un puntero temporal (temp) para guardar la referencia al nodo actual.

Finalmente, se llama a delete temp; para liberar la memoria de cada Nodo4<Punto>. Al hacer esto en el bucle anidado, se asegura que todos los nodos, fila por fila, sean borrados correctamente.

Liberación de las líneas: A diferencia de la malla, las Lineas están almacenadas en una ListaEnlazada. El destructor recorre esta lista.

Un puntero de línea (lineaPtr) se inicializa en el primer nodo de la lista con lineas.obtenerCabeza().

El bucle (while (lineaPtr)) avanza por la lista, moviendo el puntero hacia la derecha con lineaPtr->obtenerDerecha().

Dentro del bucle, se usa delete lineaPtr->obtenerDato(); para liberar la memoria del objeto Linea* almacenado dentro de cada nodo.

Es importante notar que el destructor no libera los nodos de la lista enlazada, ya que el destructor de la ListaEnlazada se encarga de esa tarea automáticamente. El destructor del tablero solo se ocupa de los objetos de Linea que fueron creados dinámicamente.

4. Elementos de Programación
El destructor de Tablero utiliza los siguientes elementos de C++:
Punteros (*) y delete: El uso de punteros y el operador delete es fundamental para la gestión manual de la memoria, una característica clave de C++.
Bucles while: La lógica de recorrido de las estructuras se implementa con bucles while, lo que permite iterar a través de las mallas y listas de tamaño variable hasta llegar a un puntero nullptr.

CrearMalla( )
1. Descripción General
El método crearMalla() es el corazón de la lógica de inicialización de la clase Tablero. Su único propósito es construir dinámicamente una cuadrícula bidimensional de objetos Punto a partir de una serie de nodos Nodo4 interconectados. El método orquesta la creación de todos los puntos en el tablero y establece los enlaces bidireccionales verticales y horizontales entre ellos. Este proceso es fundamental para la estructura del tablero y para la posterior navegación por la cuadrícula para gestionar líneas y comprobar las celdas completadas.

2. Funcionalidad Clave
Las funciones clave de este método son:

Instanciación de Puntos: Crea un objeto Punto para cada coordenada en el tablero, desde (0, 0) hasta (filas-1, columnas-1).

Construcción de la cuadrícula: Enlaza los objetos Punto entre sí usando nodos Nodo4 para formar una malla conectada, lista para ser recorrida en las cuatro direcciones.

Inicialización del puntero inicio: Establece la variable miembro inicio para que apunte al primer Nodo4<Punto> de la cuadrícula, proporcionando un único punto de entrada para todas las operaciones del tablero.

3. Mecanismos y Lógica (Paso a Paso)
El método crearMalla utiliza un enfoque de varias etapas para construir la cuadrícula. Aquí tienes un recorrido detallado de su lógica:

Paso 1: Creación de Nodos y Enlaces Horizontales

Esta primera etapa crea cada fila de la cuadrícula como una lista enlazada separada.
El método utiliza una ListaEnlazada temporal para contener las listas de puntos, que llamaremos filasLista.

Un bucle for anidado itera a través de cada fila (f) y cada columna (c) del tablero.

Para cada fila f, se crea una nueva ListaEnlazada<Punto> llamada filaActual.

Para cada columna c de esa fila, se crea un objeto Punto con las coordenadas (f, c).

Se llama al método filaActual->insertarFinal() para cada punto, que se encarga de crear un nuevo Nodo4<Punto> y establece los enlaces horizontales (derecha e izquierda) dentro de esa fila.

Una vez que se crea una fila completa, se llama a filasLista.insertarFinal(filaActual), que coloca el puntero a la lista de la fila recién creada dentro de otro Nodo4. Al final de este paso, tenemos una lista de listas, donde cada lista interna es una fila de puntos enlazada horizontalmente.

Paso 2: Enlace Vertical de Filas
Esta es la parte más crítica del método, donde las filas se conectan para formar una cuadrícula cohesiva.

El método obtiene dos punteros: nodoFila (para la fila actual) y nodoFilaAbajo (para la fila debajo). Estos punteros se utilizan para recorrer la lista principal de filas.

Un bucle while itera mientras ambos punteros sean válidos (es decir, no nullptr).

Dentro de este bucle principal, se utiliza otro bucle while anidado para iterar horizontalmente a través de los puntos tanto de la fila actual como de la que está debajo.

Dos nuevos punteros, nodoCol y nodoColAbajo, se inicializan en el primer punto de la fila actual y en el primer punto de la fila de abajo, respectivamente.

El bucle while interno se ejecuta, realizando las siguientes acciones clave en cada par de nodos (uno de cada fila):

nodoCol->establecerAbajo(nodoColAbajo): Establece el puntero abajo del nodo actual para que apunte al nodo de la fila inferior.

nodoColAbajo->establecerArriba(nodoCol): Establece el puntero arriba del nodo de la fila de abajo para que apunte de nuevo al nodo actual. Esto crea un enlace vertical bidireccional.

Luego, el bucle avanza ambos punteros horizontalmente (nodoCol = nodoCol->obtenerDerecha() y nodoColAbajo = nodoColAbajo->obtenerDerecha()) para procesar el siguiente par de nodos en las columnas.

Una vez que el bucle interno termina (todas las columnas de una fila están enlazadas), el bucle exterior avanza a la siguiente fila (nodoFila = nodoFilaAbajo) y actualiza el puntero nodoFilaAbajo, preparándose para el siguiente pase de enlace vertical.

Paso 3: Asignación del Puntero inicio
Después de que los bucles se completan y toda la cuadrícula está enlazada, el método establece la variable miembro inicio de la clase Tablero.

El inicio apunta al primer nodo de la cuadrícula, que se encuentra al acceder a la cabeza de filasLista y luego obtener la cabeza de la lista de puntos de la primera fila. Esto asegura que se pueda acceder y recorrer toda la cuadrícula a partir de un único punto bien definido.

4. Elementos de Programación
Este método utiliza los siguientes elementos de C++:

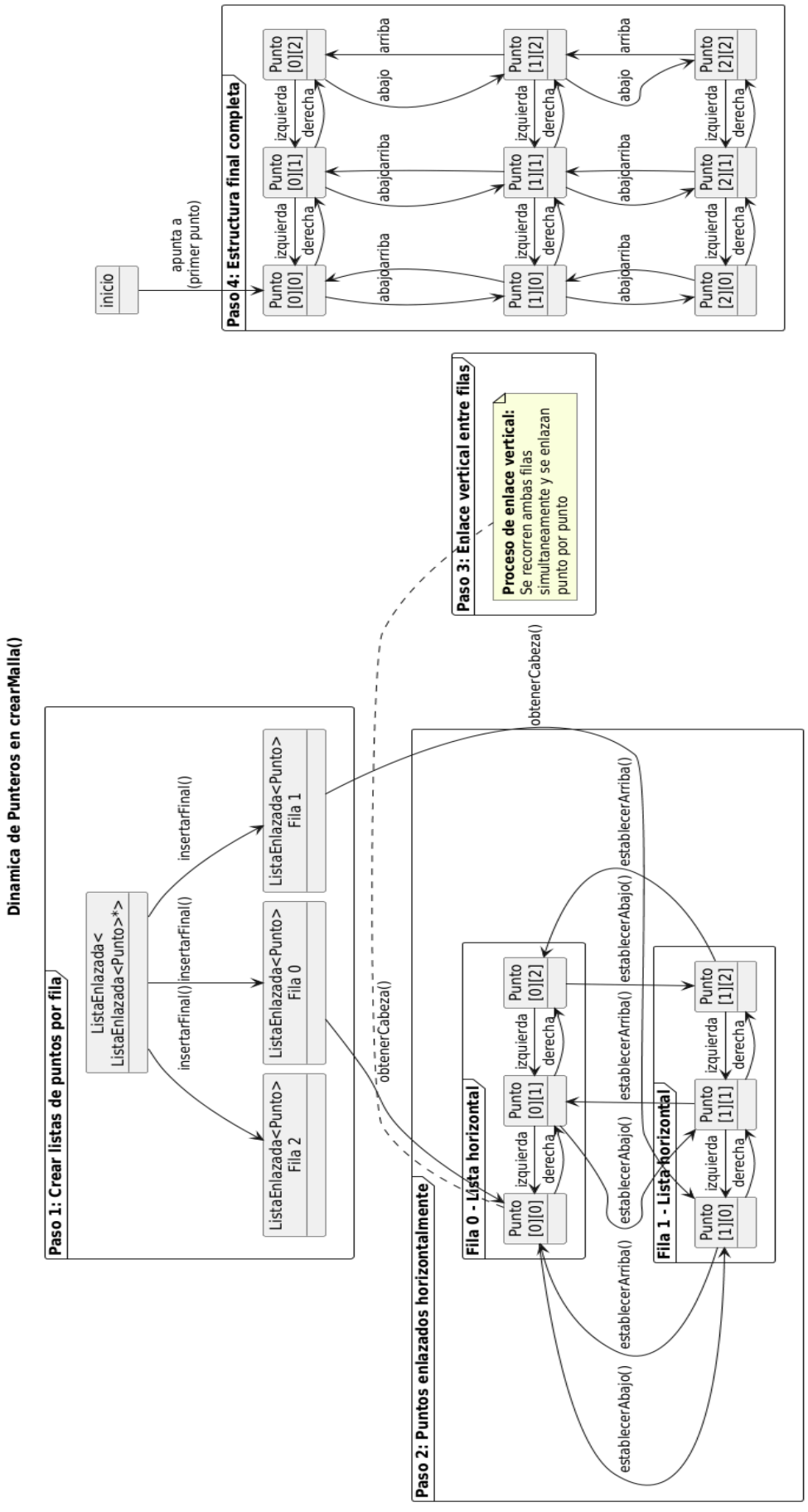
Sentencias if y for: Para el control de flujo y la creación de la estructura inicial del tablero.

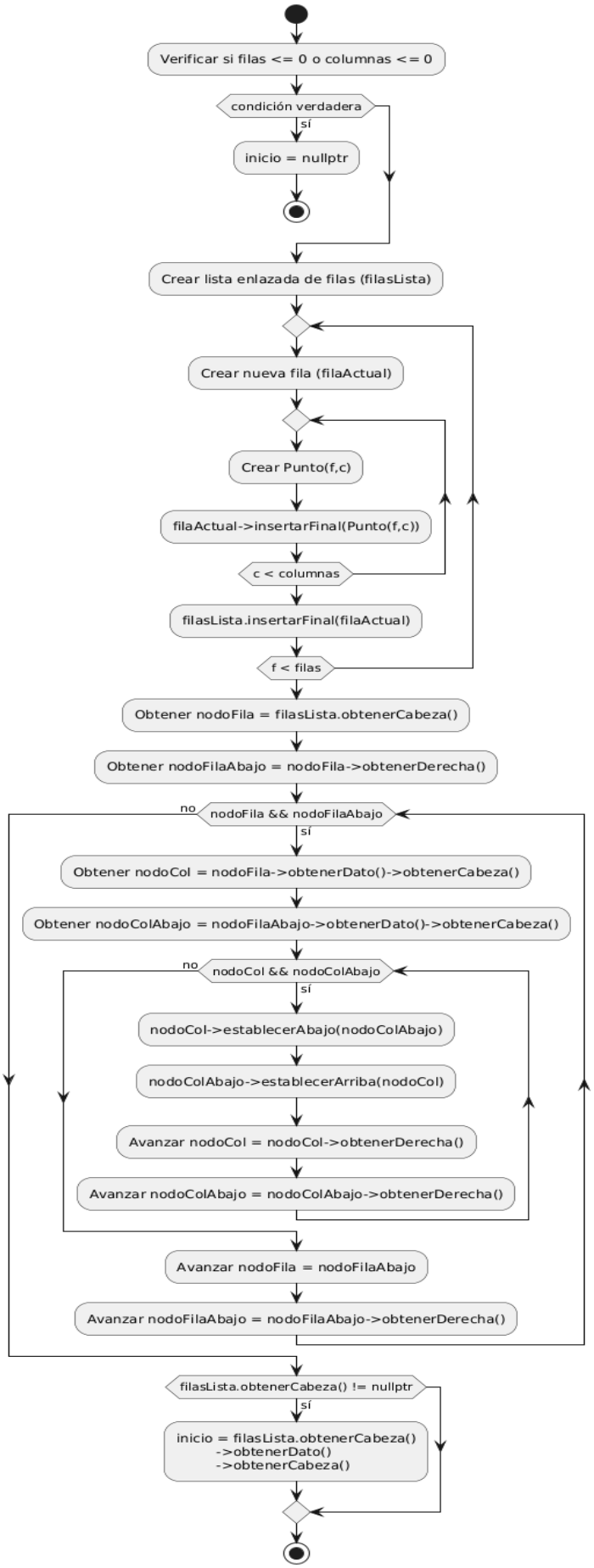
Bucles while: Para recorrer las filas y columnas con el fin de establecer los complejos enlaces verticales.

Punteros y new: El método utiliza ampliamente los punteros a Nodo4 para crear y enlazar los nodos de la cuadrícula en la memoria.

Clases relacionadas: Se basa en gran medida en las clases Punto, Nodo4 y ListaEnlazada para realizar sus tareas. La capacidad de la clase Nodo4 para enlazar en cuatro direcciones es lo que hace posible esta estructura de cuadrícula.

Inyección de Dependencias: Aunque el método crea Puntos, se basa en la estructura preexistente de Nodo4 para construir la cuadrícula.


Dinamica de Punteros en crearMalla()
Paso 1: Crear listas de puntos por fila
ListaEnlazada<
ListaEnlazada<Punto>*>
insertarFinal()
insertarFinal()
insertarFinal()
ListaEnlazada<Punto>
Fila 2
ListaEnlazada<Punto>
Fila 0
ListaEnlazada<Punto>
Fila 1
obtenerCabeza()
obtenerCabeza()
Paso 3: Enlace vertical entre filas
Proceso de enlace vertical:
Se recorren ambas filas
simultaneamente y se enlazan
punto por punto
Paso 2: Puntos enlazados horizontalmente
Fila 0 - Lista horizontal
Punto
[0][0]
izquierda
derecha
Punto
[0][1]
Punto
[0][2]
Fila 1 - Lista horizontal
Punto
[1][0]
Punto
[1][1]
Punto
[1][2]
establecerAbajo()
establecerArriba()
establecerAbajo()
establecerArriba()
establecerArriba()
establecerAbajo()
inicio
apunta a
(primer punto)
Paso 4: Estructura final completa
Punto
[0][0]
Punto
[0][1]
Punto
[0][2]
Punto
[1][0]
Punto
[1][1]
Punto
[1][2]
Punto
[2][0]
Punto
[2][1]
Punto
[2][2]
izquierda
derecha
abajo
arriba
abajo
arriba
abajo
arriba
abajo
arriba


Verificar si filas <= 0 o columnas <= 0
condición verdadera
sí
inicio = nullptr
Crear lista enlazada de filas (filasLista)
Crear nueva fila (filaActual)
Crear Punto(f,c)
filaActual->insertarFinal(Punto(f,c))
c < columnas
filasLista.insertarFinal(filaActual)
f < filas
Obtener nodoFila = filasLista.obtenerCabeza()
Obtener nodoFilaAbajo = nodoFila->obtenerDerecha()
no
nodoFila && nodoFilaAbajo
sí
Obtener nodoCol = nodoFila->obtenerDato()->obtenerCabeza()
Obtener nodoColAbajo = nodoFilaAbajo->obtenerDato()->obtenerCabeza()
no
nodoCol && nodoColAbajo
sí
nodoCol->establecerAbajo(nodoColAbajo)
nodoColAbajo->establecerArriba(nodoCol)
Avanzar nodoCol = nodoCol->obtenerDerecha()
Avanzar nodoColAbajo = nodoColAbajo->obtenerDerecha()
Avanzar nodoFila = nodoFilaAbajo
Avanzar nodoFilaAbajo = nodoFilaAbajo->obtenerDerecha()
filasLista.obtenerCabeza() != nullptr
sí
inicio = filasLista.obtenerCabeza()
->obtenerDato()
->obtenerCabeza()

generarLineas( )

1. Descripción General

El método generarLineas() es el responsable de crear todos los objetos Linea que conforman el tablero del juego. Su propósito es recorrer la malla de Puntos previamente construida y, por cada Punto, generar y almacenar las líneas que lo conectan con sus vecinos adyacentes (a la derecha y abajo). Este proceso es fundamental para inicializar el juego, ya que las líneas son los elementos que los jugadores colocarán para capturar celdas.

2. Funcionalidad Clave

Las funciones principales de este método son:

Creación de líneas: Crea dinámicamente cada objeto Linea en el tablero.

Establecimiento de orientación: Asigna la orientación correcta (HORIZONTAL o VERTICAL) a cada línea.

Almacenamiento: Agrega todas las líneas creadas a la ListaEnlazada<Linea*> de la clase Tablero, lo que permite su posterior gestión por el GestorLineas.

3. Mecanismos y Lógica (Paso a Paso)

La lógica de generarLineas() se basa en un recorrido sistemático de la cuadrícula de puntos. Utiliza una estrategia de "barrido" para asegurar que cada posible línea se cree una sola vez.

Recorrido de la Malla: El método usa un bucle anidado para iterar a través de la malla de Puntos, de manera similar a cómo se construyó.

Un puntero de fila, filaPtr, se inicializa en el inicio del tablero y se mueve hacia abajo en cada iteración del bucle externo.

Un puntero de columna, colPtr, se inicializa en el inicio de cada fila y se mueve hacia la derecha en cada iteración del bucle interno.

Verificación y Creación de Líneas: Dentro del bucle anidado, el método realiza dos verificaciones clave en cada Punto para determinar si puede crear una línea:

Línea horizontal: Comprueba si el punto actual tiene un vecino a su derecha (colPtr->obtenerDerecha()). Si lo tiene, significa que puede existir una línea horizontal. Se crea un nuevo objeto Linea y se le pasan los punteros a los dos puntos (colPtr->obtenerDato() y el punto a su derecha) junto con la Orientacion::HORIZONTAL. La línea recién creada se inserta al final de la lineas del Tablero.

Línea vertical: De forma similar, comprueba si el punto actual tiene un vecino abajo (colPtr->obtenerAbajo()). Si lo tiene, se crea una nueva Linea vertical, se le pasan los punteros a los dos puntos y la Orientacion::VERTICAL, y se inserta en la lista de lineas.

Avance de Punteros: Después de verificar y posiblemente crear las líneas para el punto actual, colPtr avanza a la siguiente columna (colPtr->obtenerDerecha()). Cuando el bucle interno finaliza (es decir, el colPtr llega al final de la fila), el bucle externo avanza filaPtr a la siguiente fila (filaPtr->obtenerAbajo()).

Este enfoque garantiza que cada línea sea creada exactamente una vez. Por ejemplo, la línea entre Punto(0, 0) y Punto(0, 1) solo se genera cuando el colPtr está en Punto(0, 0), no cuando está en Punto(0, 1).

4. Elementos de Programación
Este método utiliza los siguientes elementos de C++:

Punteros (*): Es fundamental el uso de punteros para navegar por la malla de Puntos y para crear y gestionar los objetos Linea dinámicamente.

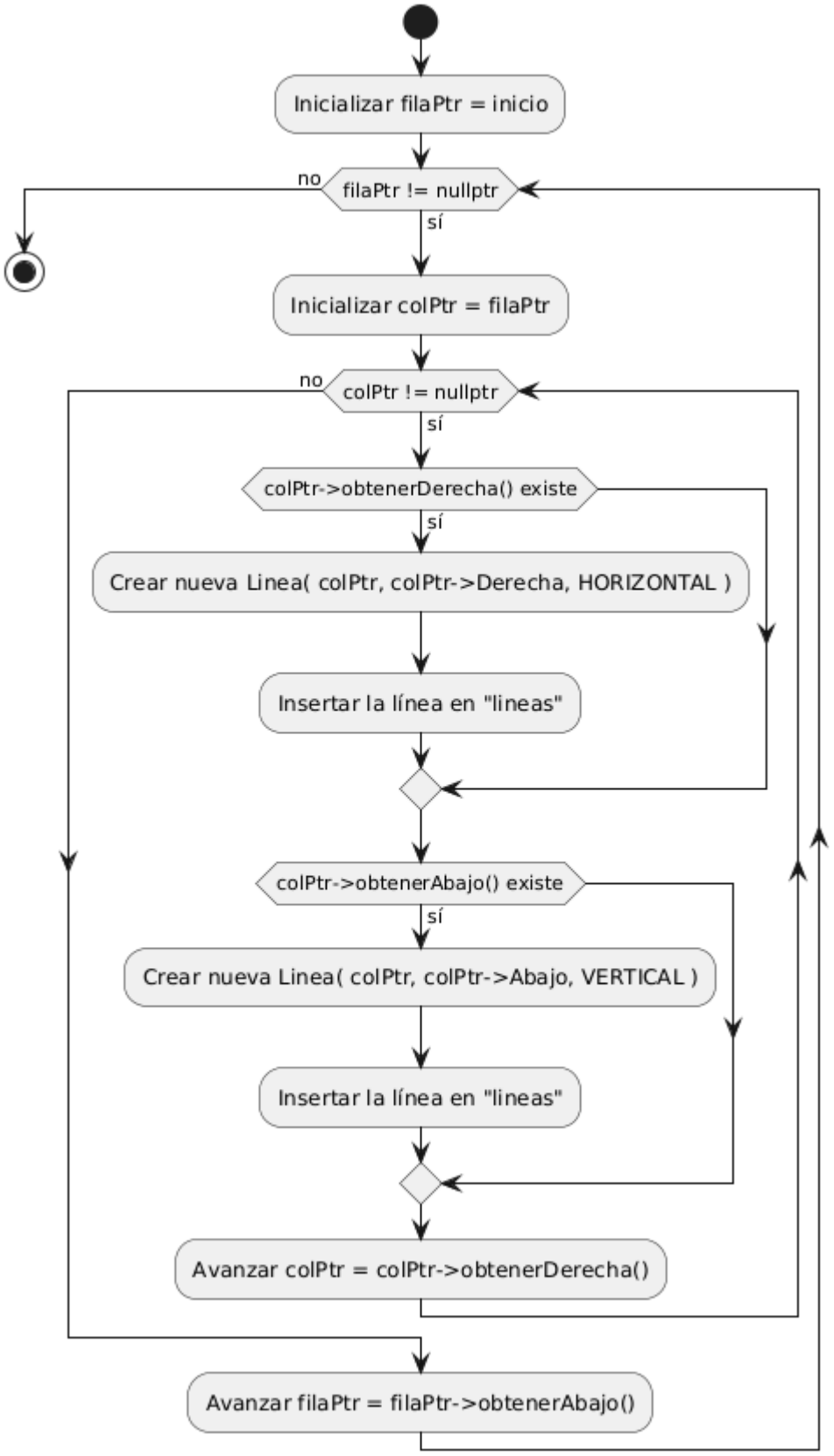
new: El operador new se utiliza para la asignación dinámica de memoria de cada objeto Linea. La ListaEnlazada del Tablero se encarga de gestionar estos punteros.

Bucles anidados while: La lógica de recorrido de la malla se implementa con bucles while, lo que permite iterar hasta el final de la estructura.

Enum class (Orientacion): El uso de la enumeración Orientacion proporciona un tipo seguro para representar la dirección de la línea, haciendo el código más legible y menos propenso a errores.

Delegación: El método delega la inserción de las líneas a la clase ListaEnlazada a través de insertarFinal(), lo que mantiene la lógica de la lista encapsulada.

Programación modular: El método generarLineas depende de crearMalla para la existencia de la cuadrícula, y a su vez, proporciona la lista de líneas al GestorLineas, mostrando una clara separación de responsabilidades.


Inicializar filaPtr = inicio
no
filaPtr != nullptr
sí
Inicializar colPtr = filaPtr
no
colPtr != nullptr
sí
colPtr->obtenerDerecha() existe
sí
Crear nueva Linea(colPtr, colPtr->Derecha, HORIZONTAL)
Insertar la línea en "lineas"
colPtr->obtenerAbajo() existe
sí
Crear nueva Linea(colPtr, colPtr->Abajo, VERTICAL)
Insertar la línea en "lineas"
Avanzar colPtr = colPtr->obtenerDerecha()
Avanzar filaPtr = filaPtr->obtenerAbajo()

generarCeldas( )

1. Descripción General

El método generarCeldas() es el encargado de construir la malla de Celdas que forma el espacio de juego. Su función principal es crear dinámicamente cada celda, asociarla con los Puntos y Lineas que la definen, y enlazarlas entre sí para formar una cuadrícula. Una característica clave es la lógica de distribución aleatoria de los PowerUps en un número predefinido de celdas, añadiendo un elemento estratégico al juego.

2. Funcionalidad Clave

Las responsabilidades de este método son:

Construcción de la malla de celdas: Crea y enlaza una cuadrícula bidireccional de objetos Celda de un tamaño (filas-1) x (columnas-1).

Asociación de componentes: Asigna los Puntos y Lineas correctos a cada Celda de la malla.

Distribución de PowerUps: Selecciona celdas al azar y asigna un PowerUp a cada una.

Gestión de memoria: Crea dinámicamente los objetos Celda y PowerUp y los enlaza a la estructura del tablero.

3. Mecanismos y Lógica (Paso a Paso)

El método utiliza múltiples bucles y un sistema de verificación para construir y poblar la malla de celdas de forma precisa.

Verificación de Precondiciones: El método comienza verificando que las dimensiones del tablero sean suficientes para formar al menos una celda (filas < 2 || columnas < 2). Si no es así, no se ejecuta ninguna lógica.

Generación de Coordenadas para PowerUps:

Se inicializa un generador de números aleatorios (srand(time(nullptr))).

Se define la cantidad fija de PowerUps a distribuir (celdasConPowerUp = 3).

Se utiliza un bucle for para generar las coordenadas de (fila, columna) para cada PowerUp.

Dentro de este bucle, un bucle do-while garantiza que no se repitan las coordenadas de las celdas elegidas. Genera una coordenada y la compara con las ya generadas; si se repite, genera una nueva.

Recorrido del Tablero de Puntos:

Se utilizan dos bucles for anidados para iterar por la cuadrícula de puntos, deteniéndose una fila y una columna antes del final (f < filas - 1, c < columnas - 1). Esto se hace porque cada celda está formada por cuatro puntos, y el bucle debe detenerse antes de llegar a los bordes.

Dentro del bucle, se obtienen los cuatro Puntos que forman la celda actual: superior izquierda, superior derecha, inferior izquierda y inferior derecha. Se hace esto navegando por los nodos colPunto y filaPunto.

Búsqueda y Asignación de Líneas:

Para cada celda, se inicializan cuatro punteros a Linea como nullptr.

Se recorre la lista completa de lineas del tablero. Por cada Linea en la lista, se llama al método l->conecta() para determinar si esa línea corresponde a uno de los cuatro lados de la celda actual. Por ejemplo, si una línea conecta el punto superior izquierdo y el superior derecho, se asigna al puntero arriba.

Una vez que se encuentran las cuatro líneas, se crean punteros a los objetos Linea correspondientes.

Creación e Inserción de la Celda:

Se crea una nueva instancia de Celda con los cuatro Puntos y los cuatro Lineas previamente encontrados.

Se crea un nuevo Nodo4<Celda> para almacenar esta Celda y se enlaza con el resto de la malla de celdas que se está construyendo.

Distribución de PowerUps:

En este punto, el algoritmo verifica si la celda actual (f, c) coincide con alguna de las coordenadas aleatorias generadas en el paso 2.

Si hay una coincidencia, se elige un tipo de PowerUp (DobleLinea, NuevasTierras, o Explosivo) al azar, se crea una nueva instancia de PowerUp y se le asigna a la celda utilizando el método asignarPowerUp(). Un mensaje por consola notifica al usuario dónde se ha colocado el PowerUp.

Enlace de la Malla de Celdas:

Enlace Horizontal: Si existe una celda anterior en la misma fila (nodoCeldaAnterior), se establecen los enlaces derecha e izquierda con el nuevo nodo de celda.

Enlace Vertical: Si existe una fila de celdas anterior (filaCeldaAnterior), se encuentra el nodo de celda que está justo encima de la celda actual y se establecen los enlaces arriba y abajo entre ellos.

Los punteros de recorrido (nodoCeldaAnterior, filaCeldaAnterior, colPunto, filaPunto) se actualizan para el siguiente ciclo de los bucles.

4. Elementos de Programación
Este método utiliza una amplia gama de elementos de C++:

Bucles anidados for y while: Para recorrer las estructuras de datos y realizar las operaciones.

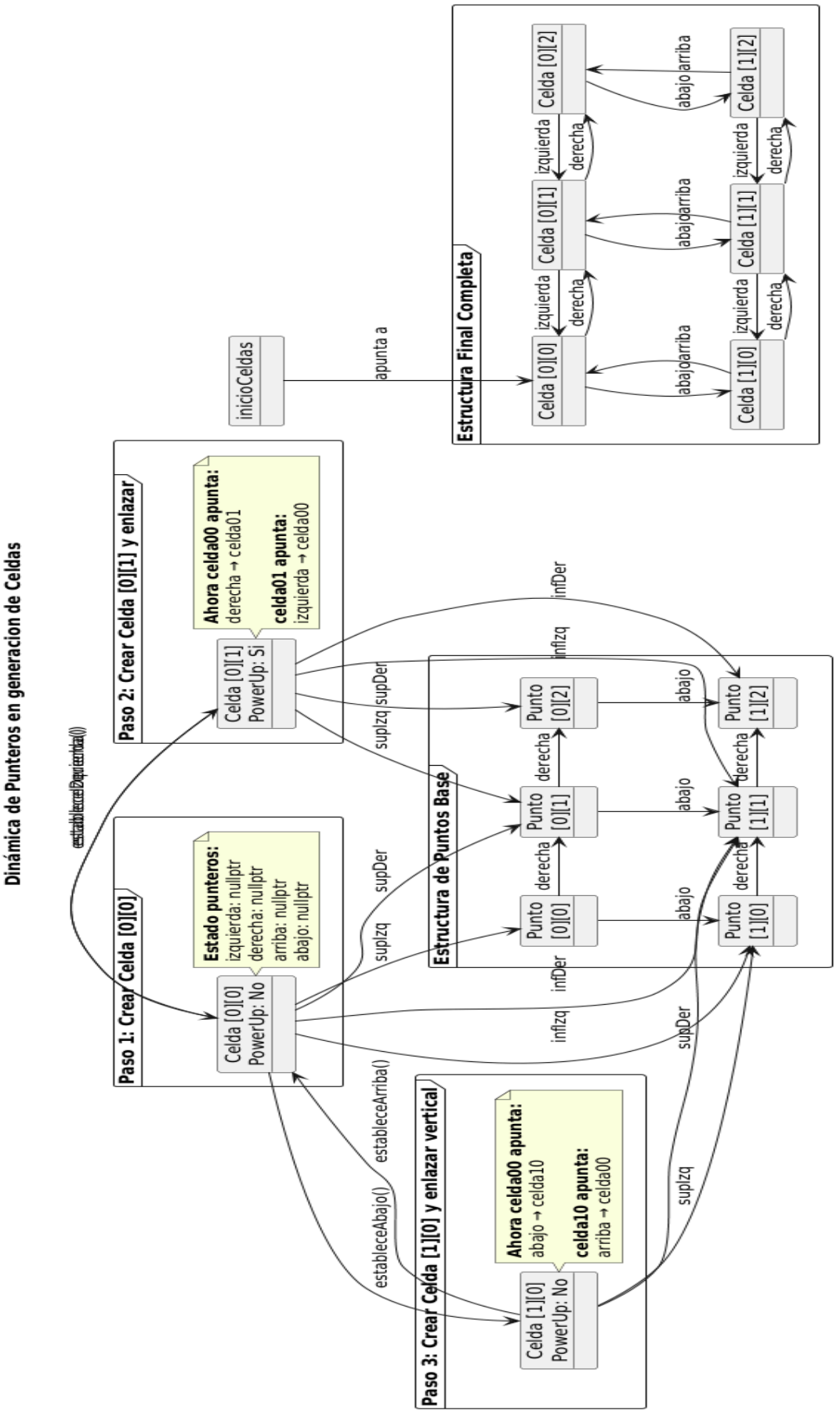
Punteros y new: Creación dinámica y manejo de punteros a Punto, Linea, Celda, y PowerUp.

Condicionales (if/else): Para verificar condiciones, como la validez de los parámetros, la existencia de PowerUps o la orientación de las líneas.

Generación de números aleatorios (rand, srand): Para la distribución de PowerUps.

Inyección de dependencias: La clase Celda es creada con punteros a Puntos y Lineas, lo que demuestra que generarCeldas utiliza datos de otras partes del tablero sin poseerlos.

Polimorfismo: La creación de PowerUps utiliza un puntero de la clase base (PowerUp*) que puede apuntar a objetos de las clases derivadas (DobleLinea, NuevasTierras, Explosivo), una característica clave de la programación orientada a objetos.


Dinámica de Punteros en generacion de Celdas
Paso 1: Crear Celda [0][0]
Celda [0][0]
PowerUp: No
Estado punteros:
izquierda: nullptr
derecha: nullptr
arriba: nullptr
abajo: nullptr
Paso 2: Crear Celda [0][1] y enlazar
Celda [0][1]
PowerUp: Si
Ahora celda00 apunta:
derecha → celda01
celda01 apunta:
izquierda → celda00
Paso 3: Crear Celda [1][0] y enlazar vertical
Celda [1][0]
PowerUp: No
Ahora celda00 apunta:
abajo → celda10
celda10 apunta:
arriba → celda00
establecerDerecha()
establecerIzquierda()
establecerArriba()
establecerAbajo()
supDer
supIzq
infDer
infIzq
Estructura de Puntos Base
Punto [0][0]
Punto [0][1]
Punto [0][2]
Punto [1][0]
Punto [1][1]
Punto [1][2]
derecha
abajo
inicioCeldas
apunta a
Estructura Final Completa
Celda [0][0]
Celda [0][1]
Celda [0][2]
Celda [1][0]
Celda [1][1]
Celda [1][2]
izquierda
derecha
abajo
arriba

Expandir nuevas tierras

1. Descripción General

El método expandirAbajo() es el encargado de aumentar el tamaño del tablero agregando una nueva fila en su parte inferior. Su objetivo principal es extender la cuadrícula de Puntos y la malla de Celdas, creando los nuevos nodos y estableciendo todos los enlaces necesarios para que la nueva fila quede perfectamente integrada a la estructura existente. Esta funcionalidad es la base del PowerUp "Nuevas Tierras" y demuestra la naturaleza dinámica del tablero.

2. Funcionalidad Clave

Las responsabilidades de este método son:
Expansión de la malla de puntos: Añade una nueva fila de Puntos en la parte inferior del tablero.
Integración de enlaces: Conecta la nueva fila de Puntos con la fila de Puntos anterior, asegurando la navegabilidad vertical.
Expansión de la malla de celdas: Crea una nueva fila de Celdas encima de la nueva fila de puntos.
Creación de nuevas líneas: Genera las líneas horizontales y verticales necesarias para las nuevas celdas.

3. Mecanismos y Lógica (Paso a Paso)

El método expandirAbajo() se ejecuta en dos fases principales: primero, expande la malla de Puntos, y luego, la malla de Celdas. La clave del éxito es la manipulación precisa de los punteros para establecer los enlaces correctos.

Paso 1: Expansión y Enlace de la Malla de Puntos

Encontrar la última fila: El algoritmo comienza recorriendo el tablero verticalmente, desde inicio, hasta encontrar la última fila de puntos. Lo hace navegando con el puntero ultimaFila hacia abajo (ultimaFila = ultimaFila->obtenerAbajo()) hasta que ultimaFila->obtenerAbajo() es nullptr.

Crear la nueva fila: Se inicia un bucle for para crear una nueva fila de Puntos del mismo ancho (columnas) que el tablero.

Enlaces horizontales de la nueva fila: En cada iteración del bucle, se crea un nuevo Nodo4<Punto>. Si es el primer punto de la nueva fila, se convierte en el nuevaFila. Si no, se enlaza con el punto anterior de la nueva fila, estableciendo un enlace horizontal bidireccional (currentNew->establecerDerecha(nuevoNodo) y nuevoNodo->establecerIzquierda(currentNew)).

Enlaces verticales de la nueva fila: En la misma iteración del bucle, se toma el Punto de la última fila original (currentOld) que está justo encima del nuevo Punto. Se establecen los enlaces verticales entre estos dos puntos (currentOld->establecerAbajo(nuevoNodo) y nuevoNodo->establecerArriba(currentOld)). Esto conecta la nueva fila con el resto del tablero.

Avance de punteros: Los punteros currentNew y currentOld avanzan a sus siguientes puntos, asegurando que el proceso de enlace continúe de manera coordinada.

Actualización de dimensiones: Al finalizar el bucle, la variable filas se incrementa en uno para reflejar el nuevo tamaño del tablero.

Paso 2: Expansión y Enlace de la Malla de Celdas

Encontrar la última fila de celdas: De forma similar, el algoritmo encuentra la última fila de Celdas existente (ultimaFilaCeldas) navegando hacia abajo desde inicioCeldas.

Crear la nueva fila de celdas: Se inicia un bucle for para crear una nueva fila de celdas. Esta fila tendrá columnas-1 celdas, ya que las celdas tienen una dimensión menor que la malla de puntos.

Identificar componentes de la celda: Para cada nueva celda, se identifican los cuatro puntos que la forman. Los puntos superiores provienen de la ultimaFila de puntos, mientras que los puntos inferiores provienen de la nuevaFila de puntos que se acaba de crear.

Buscar y crear líneas: Se buscan las líneas horizontales y verticales de la celda superior e izquierda, ya que estas ya existen. Sin embargo, las líneas horizontales y verticales que conectan los puntos de la nueva fila deben ser creadas. Se usa new Linea() para estas nuevas líneas y se insertan en la lista de líneas del tablero.

Creación y enlace de la nueva celda:

Se crea un nuevo objeto Celda, pasándole los punteros a los puntos y líneas que lo componen.

Se crea un nuevo Nodo4<Celda> para esta celda.

Enlace horizontal: Se enlaza horizontalmente con la celda anterior de la nueva fila, si existe. Esto se hace con establecerDerecha y establecerIzquierda.

Enlace vertical: Se busca la celda que está justo encima de la nueva celda en la última fila de celdas (ultimaFilaCeldas) y se establecen los enlaces bidireccionales establecerAbajo y establecerArriba.

Avance de punteros: Los punteros puntoSuperior y puntoInferior se mueven a sus siguientes puntos, y los punteros de la nueva fila de celdas (currentCeldaNew) también avanzan.

4. Elementos de Programación
Este método demuestra una compleja manipulación de punteros en C++:

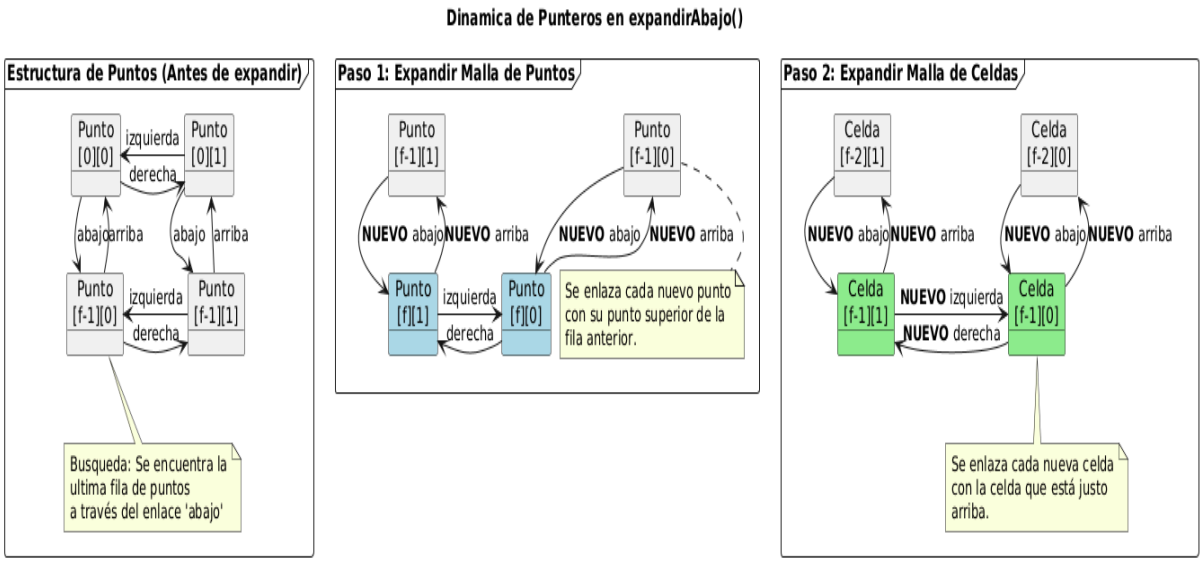
Punteros (*) y new: La creación y gestión manual de punteros es fundamental para la expansión de la estructura de datos.

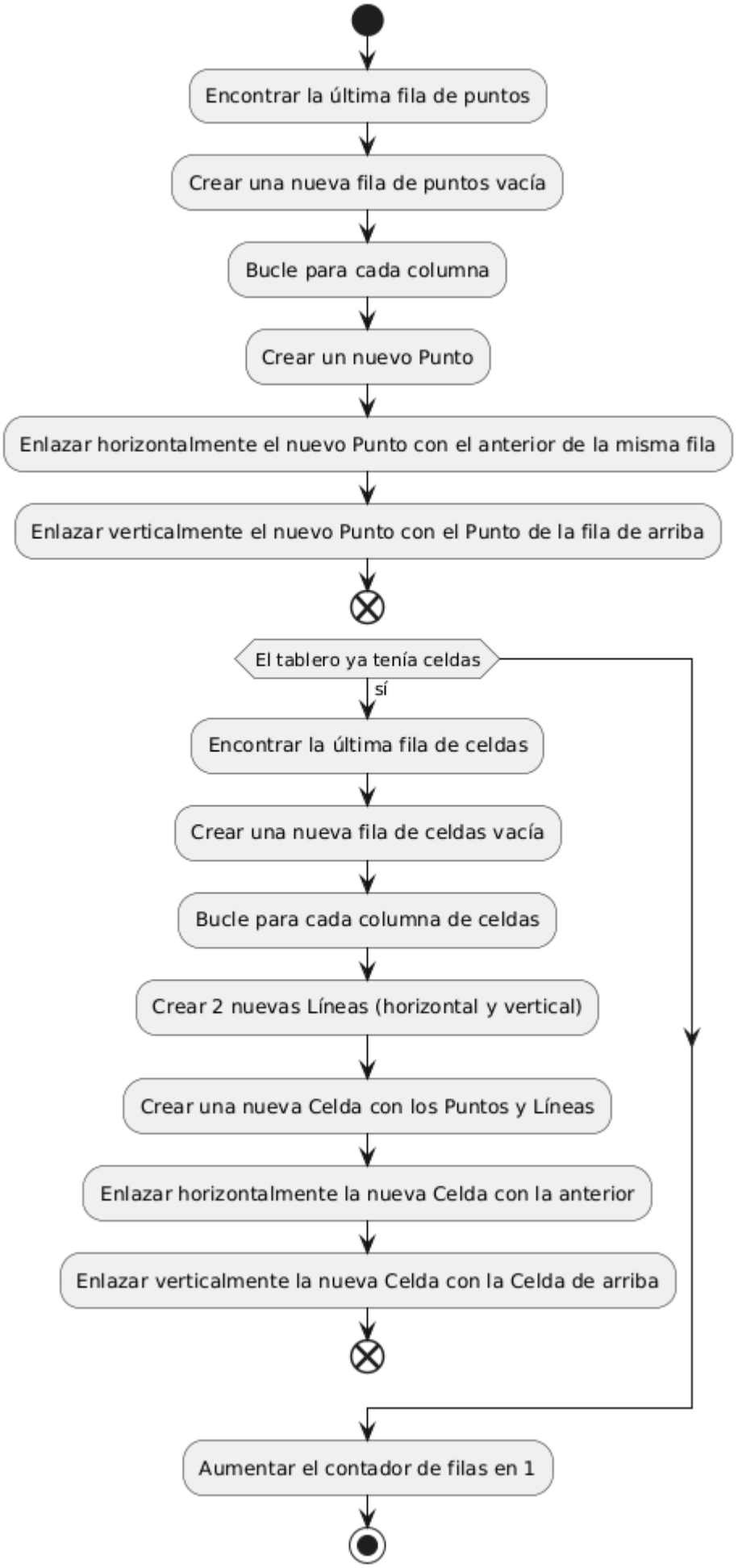
Bucles while y for: Se utilizan para el recorrido y la construcción de las nuevas filas y celdas.

const y nullptr: Se usan para garantizar que los punteros sean válidos y que las referencias a los datos no se alteren.

Delegación: El método delega la búsqueda de líneas existentes a un método buscarLinea y la inserción de nuevas líneas a la ListaEnlazada.

Asociación y Composición: Muestra una asociación con Punto y Linea, y una composición (a través de new) al crear las nuevas celdas.




Encontrar la última fila de puntos
Crear una nueva fila de puntos vacía
Bucle para cada columna
Crear un nuevo Punto
Enlazar horizontalmente el nuevo Punto con el anterior de la misma fila
Enlazar verticalmente el nuevo Punto con el Punto de la fila de arriba
El tablero ya tenía celdas
sí
Encontrar la última fila de celdas
Crear una nueva fila de celdas vacía
Bucle para cada columna de celdas
Crear 2 nuevas Líneas (horizontal y vertical)
Crear una nueva Celda con los Puntos y Líneas
Enlazar horizontalmente la nueva Celda con la anterior
Enlazar verticalmente la nueva Celda con la Celda de arriba
Aumentar el contador de filas en 1

Juego

1. Descripción General

La clase Juego es el director y controlador principal del juego. Encapsula toda la lógica de la partida, desde la inicialización del tablero y los jugadores hasta el bucle principal del juego y la gestión de turnos. Su propósito es coordinar las interacciones entre todas las demás clases (Tablero, GestorLineas, Jugador, PowerUp, etc.) y gestionar el flujo de la partida de principio a fin.

2. Funcionalidad Clave

Juego es la clase central que orquesta la partida. Sus funciones principales son:

Orquestación: Controla la secuencia del juego, incluyendo la gestión de turnos, la validación de movimientos y la detección de la finalización del juego.

Integración de Clases: Actúa como un puente entre la lógica del juego y las estructuras de datos, conectando Configuracion, Tablero, GestorLineas y Jugador.

Manejo de Turnos: Utiliza una Cola para gestionar el orden de los jugadores, siguiendo el principio de "primero en entrar, primero en salir" (FIFO).

Interacción con el usuario: Maneja la entrada y salida de datos a través de la consola, solicitando movimientos y mostrando el estado del juego.

Lógica de PowerUps: Proporciona los métodos para que los PowerUps interactúen con el tablero y otros elementos del juego, demostrando polimorfismo.

Finalización del juego: Determina cuándo la partida ha terminado y evalúa al ganador basándose en la puntuación.

3. Mecanismos y Lógica

La lógica de Juego está construida alrededor del bucle principal (jugar()), que se ejecuta hasta que se cumple una condición de finalización.

Constructor y Destructor: El constructor inicializa los punteros a nullptr y el destructor se encarga de liberar la memoria de tablero y gestor, evitando fugas de memoria.

inicializar(): Este método es la puerta de entrada a la partida. Crea una nueva instancia de Tablero y GestorLineas basándose en la Configuracion proporcionada. Luego, encola a todos los jugadores en la colaTurnos para establecer el orden inicial.

jugar() (Bucle principal): Este es el corazón del programa.

Obtener Turno: Desencola al siguiente Jugador para que tome su turno.

Manejo de PowerUps: Le pregunta al jugador si desea usar un PowerUp. Si el jugador responde que sí, se llama a usarPowerUp() en el objeto Jugador. La llamada polimórfica a powerUpUsado->aplicarEfecto() permite que cada tipo de PowerUp ejecute su lógica específica en el contexto del juego.

Entrada de usuario: Pide al jugador que ingrese las coordenadas de dos puntos. El método parseCoordenada() se usa para validar y convertir las entradas de string a enteros.

Colocación de línea: Se delega la lógica de colocar la línea al GestorLineas (gestor->colocarLinea()). El GestorLineas se encarga de actualizar el estado de la línea y de verificar si se han completado celdas, sumando puntos al jugador si corresponde.

Gestión de Turnos: Si no se usó un PowerUp, el Jugador actual vuelve a ser encolado al final de la cola para esperar su próximo turno.

devolverTurno(): Es un método especial que encola a un jugador en el frente de la cola, lo cual es útil para efectos de PowerUps que otorgan turnos adicionales.

Métodos de expansión y explosión: Los métodos como expandirAbajo(), expandirDerecha() y explotarPunto() son métodos delegados. En lugar de contener la lógica ellos mismos, simplemente llaman al método correspondiente en la clase Tablero, manteniendo el principio de separación de responsabilidades. Esto permite que Juego se concentre en la coordinación sin preocuparse por los detalles de implementación del tablero.

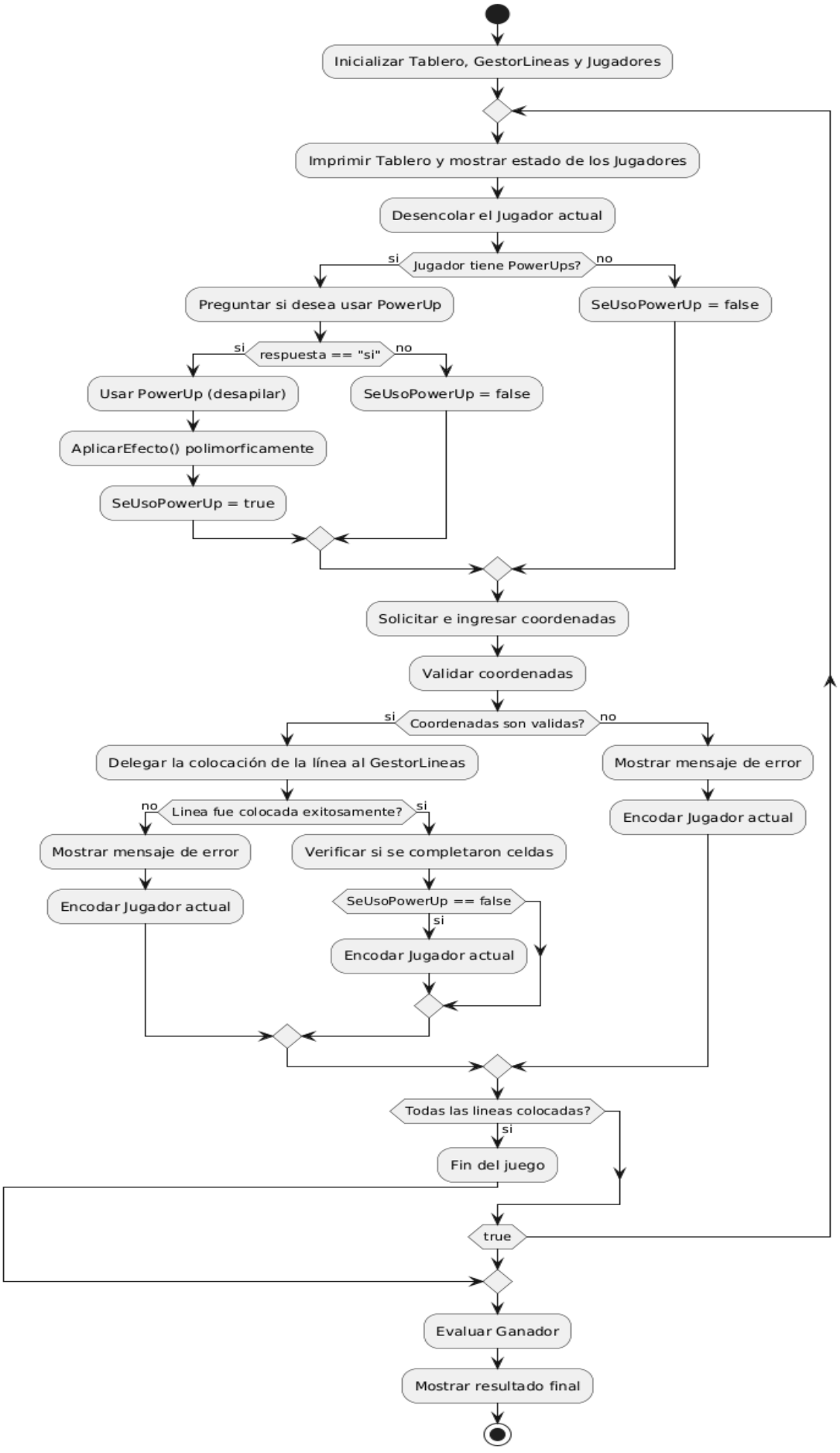
obtenerTablero(): Permite que otras clases (especialmente los PowerUps) accedan al tablero para aplicar sus efectos.

4. Elementos de Programación
Composición: La clase Juego tiene punteros a objetos de otras clases como Configuracion, Tablero, y GestorLineas. Esto demuestra una fuerte relación de composición, donde Juego "posee" y gestiona el ciclo de vida de estos objetos.

Delegación: La clase delega la mayoría de sus operaciones a sus componentes, como el manejo de líneas al GestorLineas y la manipulación del tablero a la clase Tablero.

Polimorfismo: La llamada a powerUpUsado->aplicarEfecto() es un ejemplo de polimorfismo, donde el mismo método puede tener comportamientos diferentes dependiendo del tipo de objeto (PowerUp) al que se esté llamando.

Cola<Jugador>: La estructura de datos Cola es fundamental para la gestión de turnos de los jugadores, implementando el comportamiento FIFO de forma nativa.


Inicializar Tablero, GestorLineas y Jugadores
Imprimir Tablero y mostrar estado de los Jugadores
Desencolar el Jugador actual
Jugador tiene PowerUps?
si
no
Preguntar si desea usar PowerUp
SeUsoPowerUp = false
respuesta == "si"
si
no
Usar PowerUp (desapilar)
SeUsoPowerUp = false
AplicarEfecto() polimorficamente
SeUsoPowerUp = true
Solicitar e ingresar coordenadas
Validar coordenadas
Coordenadas son validas?
si
no
Delegar la colocación de la línea al GestorLineas
Mostrar mensaje de error
Encodar Jugador actual
Linea fue colocada exitosamente?
no
si
Mostrar mensaje de error
Encodar Jugador actual
Verificar si se completaron celdas
SeUsoPowerUp == false
si
Encodar Jugador actual
Todas las lineas colocadas?
si
Fin del juego
true
Evaluar Ganador
Mostrar resultado final

PowerUp

1. Descripción General

La clase abstracta PowerUp sirve como el cimiento para todos los poderes especiales en el juego. Su propósito es definir una interfaz común para cualquier tipo de PowerUp, asegurando que todos puedan ser tratados de manera uniforme a través del polimorfismo, sin importar su función específica (como dar un turno extra o explotar un área del tablero). No puede ser instanciada por sí misma, sino que debe ser heredada por clases concretas que implementen su lógica particular.

2. Funcionalidad Clave

Esta clase establece las reglas que todos los PowerUps deben seguir:

Identificación: Todos los PowerUps tienen un simbolo de texto que los representa en la interfaz gráfica del juego.

Comportamiento Polimórfico: Define los métodos virtuales puros (virtual void... = 0;) que obligan a las clases derivadas a implementar su propia lógica. Esto permite a la clase Juego invocar un PowerUp genérico sin saber su tipo exacto.

Control de estado: Almacena si el PowerUp afecta a una línea, lo que puede ser útil para la lógica del GestorLineas o la interfaz de usuario.

3. Mecanismos y Lógica

La lógica principal de PowerUp se basa en el polimorfismo en tiempo de ejecución.

Constructor: El constructor base inicializa el simbolo y el afectaLinea de cualquier PowerUp. Esto garantiza que todas las clases derivadas comiencen con estos atributos básicos.

Destructor Virtual: El destructor es virtual (virtual ~PowerUp() {}). Esta es una práctica crucial en C++ cuando se usan clases base polimórficas. Asegura que cuando un objeto de una clase derivada es eliminado a través de un puntero a la clase base (delete powerUpUsado;), el destructor correcto de la clase derivada sea llamado primero, seguido por el destructor de la clase base. Esto previene fugas de memoria y comportamientos indefinidos.

Métodos Virtuales Puros:

activar(): Es un método abstracto. Su propósito es permitir una acción inmediata o un cambio de estado en el PowerUp en el momento de su uso, aunque en este código particular podría ser redundante ya que la mayor parte de la lógica está en aplicarEfecto. Su presencia fuerza a las clases derivadas a definir explícitamente qué sucede cuando se "activa" el poder.

aplicarEfecto(Juego* juego, Jugador* jugador): Este es el método más importante. Es virtual puro y toma punteros a Juego y Jugador. Esto permite que el PowerUp interactúe directamente con el estado de la partida y del jugador, dándole acceso a la información y los métodos necesarios para ejecutar su efecto. Por ejemplo, un PowerUp explosivo podría usar juego->explotarPunto() para afectar el tablero.

4. Elementos de Programación
Clase Abstracta: PowerUp no puede ser instanciada directamente porque contiene métodos virtuales puros. Esto la convierte en un contrato para las clases derivadas.

Punteros a clases (class Jugador;, class Juego;): Las declaraciones forward en el archivo de cabecera minimizan las dependencias. Esto permite que PowerUp tenga un puntero a Juego y Jugador sin tener que incluir sus archivos de cabecera completos, lo que reduce el tiempo de compilación.

virtual: La palabra clave virtual es la piedra angular del polimorfismo en C++. Permite que la función correcta sea llamada en tiempo de ejecución basándose en el tipo real del objeto.

const: Los getters están marcados como const para indicar que no modificarán el estado interno del objeto, lo cual es una buena práctica de diseño.

Nuevas Tierras
1. Descripción General
La clase NuevasTierras es un tipo específico de PowerUp que permite a un jugador expandir el tablero de juego. Su principal función es interactuar con el usuario para obtener un punto en el borde del tablero y, a continuación, llamar a los métodos de expansión correspondientes en la clase Juego. Este PowerUp proporciona un elemento estratégico, permitiendo a los jugadores crear nuevas oportunidades para capturar celdas y obtener una ventaja en la partida.

2. Funcionalidad Clave
Expansión Dinámica del Tablero: Su objetivo central es invocar las funciones de expansión (expandirAbajo o expandirDerecha) de la clase Juego para modificar las dimensiones del tablero durante la partida.

Validación de la Entrada: Valida la coordenada del punto que el jugador elige para asegurar que la expansión sea posible y válida.

Control de Flujo del Juego: Al final de su efecto, otorga al jugador un turno adicional llamando al método devolverTurno de la clase Juego.

3. Mecanismos y Lógica (Paso a Paso)
El método aplicarEfecto() de NuevasTierras contiene la lógica principal, que se puede dividir en varios pasos clave:

Solicitud de Coordenada: El método solicita al jugador que ingrese una coordenada de un punto en el borde derecho o inferior del tablero. Esta es la entrada que define dónde ocurrirá la expansión.

Validación de la Coordenada:

Primero, utiliza el método juego->parseCoordenada() para convertir la entrada del usuario de una string (ej. "A0") a coordenadas numéricas (fila, columna) y valida que el formato sea correcto.

Después, usa el método juego->estaEnBorde() para asegurarse de que el punto elegido esté, de hecho, en uno de los bordes o en la esquina del tablero. Si no, el PowerUp no se aplica y la ejecución termina.

Determinación de la Dirección de Expansión:

Se obtienen las dimensiones actuales del tablero (filas, columnas) desde la clase Juego.

La lógica usa una serie de condicionales if-else para determinar la dirección de la expansión basándose en la coordenada del punto elegido:

Si el punto está en la esquina inferior derecha (f1 == filas - 1 && c1 == columnas - 1), se le da al jugador la opción de expandir hacia abajo o a la derecha. Esto demuestra la flexibilidad de la clase.

Si el punto está en el borde inferior (f1 == filas - 1), la expansión es automáticamente hacia abajo.

Si el punto está en el borde derecho (c1 == columnas - 1), la expansión es automáticamente hacia la derecha.

Si el punto no cumple ninguna de estas condiciones, se muestra un mensaje de error y el PowerUp no tiene efecto.

Ejecución de la Expansión: Una vez que se ha determinado la dirección, el método llama a juego->expandirAbajo() o juego->expandirDerecha(). Esto delega la compleja lógica de modificación del tablero a la clase Juego y, a su vez, a la clase Tablero.

Otorgar Turno Extra: Finalmente, el método llama a juego->devolverTurno(jugador). Esto le da al jugador un turno adicional, lo que recompensa su uso estratégico del PowerUp y le permite capitalizar la nueva área de juego.

El método activar() está definido pero vacío, lo que demuestra que NuevasTierras no requiere una acción previa a la aplicación de su efecto.

4. Elementos de Programación
Herencia: NuevasTierras hereda de la clase base PowerUp, lo que le permite implementar el método polimórfico aplicarEfecto() y ser tratada como un PowerUp genérico.

Delegación: El método delega la mayoría de sus operaciones a la clase Juego (parseCoordenada, estaEnBorde, expandirAbajo, expandirDerecha, devolverTurno), lo que reduce su acoplamiento y mantiene su lógica centrada en la interacción con el usuario y el control del flujo, no en la manipulación del tablero.

Validación: El uso de condicionales y mensajes de error demuestra una lógica defensiva para manejar entradas inválidas del usuario y prevenir comportamientos inesperados.

Explosivo

Manual Técnico: Clase Explosivo

1. Descripción General

La clase Explosivo es una clase concreta que hereda de PowerUp. Su objetivo principal es permitir que un jugador destruya un Punto específico en el tablero, lo que resulta en la eliminación de todas las Lineas adyacentes a dicho punto. Este PowerUp representa una herramienta disruptiva en el juego, que puede usarse para romper las estructuras de los oponentes y evitar que capturen celdas.

2. Funcionalidad Clave

Destrucción de Puntos: Su función principal es invocar un método del Tablero para eliminar un punto y sus líneas asociadas.

Interacción con el Usuario: Solicita al jugador las coordenadas del punto a explotar.

Validación: Asegura que las coordenadas ingresadas por el jugador sean válidas y estén dentro de los límites del tablero antes de aplicar el efecto.

3. Mecanismos y Lógica (Paso a Paso)

El método aplicarEfecto() de la clase Explosivo contiene la lógica principal para ejecutar la acción, siguiendo una secuencia clara de pasos:

Validación de Punteros: Primero, verifica que los punteros a Juego y Jugador no sean nulos para evitar errores.

Referencia al Tablero: Obtiene una referencia al tablero de juego a través del método juego->obtenerTablero(). Esto es un ejemplo de delegación, ya que el PowerUp no conoce el tablero directamente, sino que lo solicita al Juego para interactuar con él.

Solicitud de Entrada: Muestra el tablero actual para que el jugador pueda elegir el punto a explotar. Luego, solicita las coordenadas del punto (fila y columna) al usuario.

Validación de Coordenadas: Verifica si las coordenadas ingresadas están dentro del rango de las dimensiones del tablero (fila < tablero->getFilas() y columna < tablero->getColumnas()). Si las coordenadas son inválidas, muestra un mensaje de error y el efecto no se aplica.

Ejecución del Efecto: Si las coordenadas son válidas, delega la acción de destrucción al Tablero llamando a tablero->explotarPunto(fila, columna). Es en el Tablero donde se encuentra la lógica real para eliminar las líneas y puntos.

Manejo de Errores: Si el puntero al Tablero es nulo o las coordenadas son incorrectas, muestra mensajes de error descriptivos al usuario para guiarlo.

Destructor: El destructor de la clase Explosivo está vacío. Esto es una decisión de diseño intencional. La clase Juego elimina un PowerUp después de que se usa, pero si el destructor de Explosivo intentara manipular los nodos del tablero, podría causar un error de doble eliminación, ya que los nodos y puntos son gestionados por la clase Tablero. Un destructor vacío garantiza que el objeto Explosivo se libere de la memoria de forma segura sin afectar las estructuras de datos persistentes del tablero.

4. Elementos de Programación
Herencia Polimórfica: Explosivo hereda de la clase base PowerUp, lo que le permite implementar el método aplicarEfecto() y ser tratado de forma genérica en el Juego.

Delegación: La clase delega la lógica de manipulación del tablero a la clase Juego y al Tablero, lo que mantiene un diseño modular y de baja cohesión.

Programación Defensiva: La clase incluye verificaciones para punteros nulos y coordenadas inválidas, lo que la hace más robusta y menos propensa a fallos.

Condiciones de Victoria
por Columnas y filas
1. Descripción General
La función filtrarPorColumnasGanadas() es una condición de victoria especializada, cuyo objetivo es identificar a los jugadores que han capturado la mayor cantidad de columnas completas en el tablero. Una columna ganada es aquella en la que un solo jugador ha completado más celdas que cualquier otro jugador. La función recorre cada columna, cuenta las celdas capturadas por cada jugador, identifica al ganador de la columna y, finalmente, determina qué jugador o jugadores han ganado el mayor número de columnas en total.

2. Funcionalidad Clave
Análisis de Columna por Columna: Recorre la malla de celdas verticalmente para evaluar cada columna del tablero de forma individual.

Conteo de Celdas: Lleva un registro de cuántas celdas ha completado cada jugador en una columna específica.

Identificación del Ganador de Columna: Determina si hay un ganador claro (un solo jugador con el máximo de celdas) para cada columna.

Puntuación por Columnas: Acumula el número total de columnas ganadas para cada jugador.

Filtrado Final: Devuelve una lista con los jugadores que tienen la mayor cantidad de columnas ganadas.

3. Mecanismos y Lógica (Paso a Paso)
El algoritmo es un proceso de múltiples etapas que combina el recorrido de estructuras de datos con lógica de conteo y comparación.

Inicialización de Contadores Globales:

Se crea una ListaEnlazada<int> llamada columnasGanadas. El tamaño de esta lista es igual al número de jugadores, y cada elemento se inicializa en cero. Este contador rastreará el número total de columnas que ha ganado cada jugador a lo largo de todo el tablero.

Recorrido por Columnas (Bucle Exterior):

Se inicia un bucle while que avanza de columna en columna utilizando el puntero columnaInicio y columnaInicio->obtenerDerecha(). Este bucle asegura que se analice cada columna de celdas del tablero.

Análisis de Celdas por Columna (Bucle Anidado):

Inicialización de Contadores Locales: Dentro del bucle de columna, se crea otra ListaEnlazada<int> llamada conteoColumna, también inicializada en ceros. Esta lista es temporal y se usa para contar las celdas que cada jugador ha completado solo en la columna actual.

Recorrido de la Columna: Un bucle while anidado, usando el puntero celdaPtr y celdaPtr->obtenerAbajo(), recorre cada celda de arriba a abajo en la columna actual.

Conteo de Celdas Completadas: En cada celda, el algoritmo verifica si está completada. Si lo está, obtiene al propietario de la celda. Luego, recorre la lista de jugadores y el conteoColumna para encontrar al propietario, incrementando el contador de celdas correspondiente para esa columna.

Evaluación del Ganador de la Columna:

Después de contar las celdas en toda la columna, el algoritmo busca al ganador de esa columna.

Se encuentra el valor maxCeldas (el número máximo de celdas completadas por un jugador).

Se cuenta cuántos jugadores tienen ese valor máximo (cantidadMaximos).

Condición de Victoria de Columna: Se considera que una columna es "ganada" solo si hay un único jugador con el número máximo de celdas completadas y ese número es mayor que cero. Esto evita empates en la columna y columnas vacías.

Actualización de Contadores Globales:

Si la columna tiene un ganador claro (cumple la condición anterior), el algoritmo recorre las listas columnasGanadas y conteoColumna una vez más.

Cuando encuentra el jugador que ganó la columna, incrementa el contador de columnas ganadas para ese jugador en la lista global columnasGanadas.

Filtrado de los Ganadores Finales:

Una vez que se han evaluado todas las columnas del tablero, el algoritmo busca el número maxColumnas en la lista columnasGanadas.

Finalmente, crea una nueva ListaEnlazada<Jugador*> llamada ganadores.

Recorre las listas jugadores y columnasGanadas una última vez, y para cada jugador cuyo contador en columnasGanadas sea igual a maxColumnas, lo añade a la lista ganadores.

Retorno de la Lista: La función devuelve la lista ganadores, que contiene a todos los jugadores que han ganado la mayor cantidad de columnas.

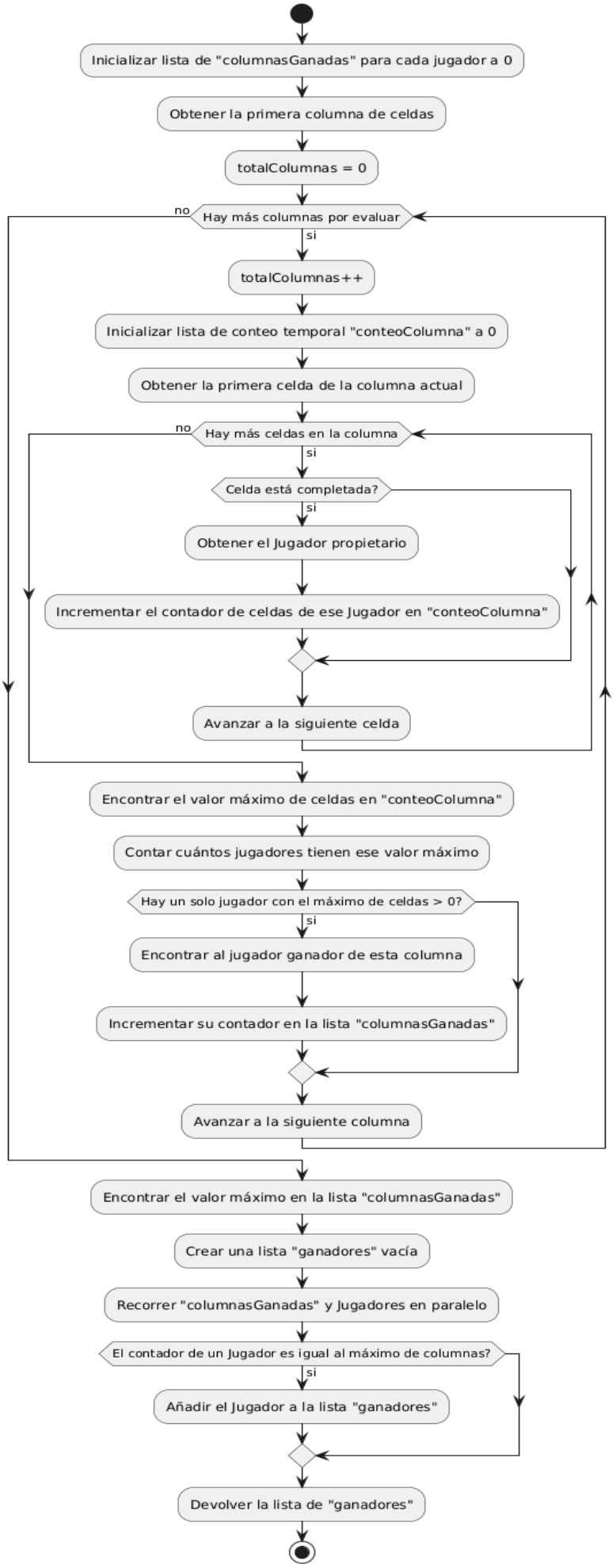
4. Elementos de Programación
Recorrido Doble (while anidados): La lógica se basa en bucles anidados para recorrer la malla de celdas de forma bidimensional (primero por columnas, luego por celdas dentro de cada columna).

Listas Enlazadas (ListaEnlazada<T>): Se utilizan extensivamente para almacenar los jugadores y sus respectivos contadores de manera dinámica, lo que es flexible para cualquier número de jugadores.

Delegación: La función depende de los métodos de Tablero (getInicioCeldas()) y Celda (estaCompletada(), getPropietario()) para acceder al estado del juego.

Variables de Estado Temporal: El uso de conteoColumna demuestra un buen manejo de variables de estado que solo son relevantes para un subproceso (una columna).

Manejo de Punteros: La manipulación cuidadosa de punteros Nodo4<T>* es fundamental para recorrer las listas y la malla sin errores.


Inicializar lista de "columnasGanadas" para cada jugador a 0
Obtener la primera columna de celdas
totalColumnas = 0
no
Hay más columnas por evaluar
si
totalColumnas++
Inicializar lista de conteo temporal "conteoColumna" a 0
Obtener la primera celda de la columna actual
no
Hay más celdas en la columna
si
Celda está completada?
si
Obtener el Jugador propietario
Incrementar el contador de celdas de ese Jugador en "conteoColumna"
Avanzar a la siguiente celda
Encontrar el valor máximo de celdas en "conteoColumna"
Contar cuántos jugadores tienen ese valor máximo
Hay un solo jugador con el máximo de celdas > 0?
si
Encontrar al jugador ganador de esta columna
Incrementar su contador en la lista "columnasGanadas"
Avanzar a la siguiente columna
Encontrar el valor máximo en la lista "columnasGanadas"
Crear una lista "ganadores" vacía
Recorrer "columnasGanadas" y Jugadores en paralelo
El contador de un Jugador es igual al máximo de columnas?
si
Añadir el Jugador a la lista "ganadores"
Devolver la lista de "ganadores"