REPLICANTES

DOCUMENTO DEL PROYECTO DE INGENIERÍA DEL SOFTWARE 2011-2012

Esta es la documentación para el proyecto Replicantes, para la asignatura de Ingeniería del software, donde se hace un resumen de las clases, métodos y diseños usados e implementados a lo largo del proyecto.

ALEXANDER MARIEL Y EGOITZ PUERTA

INTRODUCCIÓN

Nuestro juego consiste en conquistar el mayor número de casillas posibles en el tablero mediante diferentes tipos de fichas de las que dispondrán los jugadores, apropiándose de las fichas de sus enemigos cuando les se posible

He aquí un pequeño resumen más técnico:

- Comienza una partida de 2 a 4 jugadores y un tablero cuadrado de 10x10, 20x20 o 30x30.
- Se da el turno a un jugador, éste selecciona una de sus fichas.
- Selecciona que acción debe realizar su ficha: Saltar, es decir mover una ficha como en cualquier otro juego de tablero, pero a una distancia de 2 casillas exactamente.
  Replicar: Mover una ficha al igual que en otro juego de tablero, pero a una distancia de 1 casilla exactamente. Además la ficha original mantendrá su posición, es decir habremos creado una "copia" de la ficha.
- Si una ficha salta o se replica junto a una ficha enemiga la ficha enemiga es convertida y pasa a pertenecer al jugador que acaba de mover su ficha.
- Gana el jugador que más fichas tenga una vez que no se puedan realizar más movimientos sobre el tablero o el jugador que sea el último con alguna ficha sobre el tablero.

Hemos adjuntado junto a este documento un fichero .vpp que puede ser visualizado mediante el programa Visual Paradigm, el cual contiene nuestro diagrama de clases y diferentes diagramas de secuencia.

A modo de experimento comenzamos a desarrollar una variante del juego que utilizara un tablero compuesto por hexágonos, en vez de casillas cuadradas. Aun siendo nuestro diseño altamente modular, realizar el cambio de tablero de cuadrados a tablero de hexágonos implicaba modificar otras clases y métodos fuera del propio tablero en si y la interfaz gráfica.

Por esta razón decidimos dividir el proyecto en dos líneas de desarrollo distintas, una con el tablero de cuadrados que será la rama principal y la que deseamos acabar y perfeccionar al máximo. Seguimos la otra línea de desarrollo en segundo plano durante unos días para probar las ventajas del paradigma de orientación a objetos, como modularidad, abstracción, ocultación, etc. Además el tablero de hexágonos también entrañaba un reto en cuanto al diseño del tablero en sí. Finalmente esta última rama fue abandonada completamente debido a las restricciones del tiempo.

El proyecto se realizo enteramente con un diseño secuencial, con una pseudo-interfaz grafica por consola muy minimalista. Tras adquirir conocimientos sobre el manejo de las librerías Swing en clase, optamos por usarlas en nuestro proyecto para crear la interfaz gráfica. Esto nos llevó a cambiar nuestro diseño secuencial a un diseño dirigido por eventos. El cambio nos trajo numerosos problemas y tuvimos que reimplementar gran parte de lo que ya estaba completo y funcionando.

Además de una interfaz habíamos dotado a nuestro juego de una función de auto-guardado que almacenaba el progreso de la partida cada turno.

Durante la duración del proyecto nos hemos esforzado especialmente en utilizar todos los conceptos aprendidos en clase. Hemos creado una sección llamada "Librerías y patrones" donde explicamos que librerías, conceptos y patrones hemos utilizado y donde.

A continuación se detallaran las diferentes clases, con sus atributos, métodos y la funcionalidad de todos ellos en la aplicación.

LÓGICA DE NEGOCIO

1. JUGADOR

DESCRIPCIÓN BREVE:

Como su nombre bien indica esta será la clase que represente a cada jugador en la parte de la lógica de negocio.

ATRIBUTOS:

- **numJugador: int** : En este atributo guardaremos el número identificativo de este jugador. Cada jugador recibe un número único.
- **cantidadJugadores: static int** : Este atributo nos permite calcular el numero de jugadores en juego y asignar a cada nuevo jugador su número identificativo único.

MÉTODOS PRINCIPALEES:

- **equals(pJug: Jugador): boolean** : Una reimplementación mejorada del método equals() que cada clase hereda de Object. Comprueba si dos jugadores son el mismo en base al número de jugador en vez de la posición de memoria.

2. LISTAJUGADORES

DESCRIPCIÓN BREVE:

Como podemos imaginar nuestros jugadores deberán "estar" en alguna parte dentro del programa. Esta clase almacenará los jugadores que estén tomando parte en la partida en una estructura específica muy apta para el orden de los turnos.

Además de almacenar los jugadores esta clase permite comenzar una nueva partida y obtener el jugador ganador. Es decir, actúa como la clase que gestiona algunos aspectos relevantes del juego desde lo más alto de la jerarquía.

ATRIBUTOS:

- **lista: Contenedor<Jugador>** : Como podemos imaginar este es el atributo que guarda a los jugadores que estén jugando la partida. Aunque sea formal escribir "Contenedor" en vez de la estructura usada en sí aquí merece la pena hacer una excepción y definir la estructura a usar. En nuestro caso hemos decidido que una lista ligada circular es la estructura de datos perfecta para un juego como el nuestro, en el cual los turnos pasan de jugador en jugador cíclicamente de manera infinita (en realidad el recorrido infinito de la lista se detiene cuando se da una condición especial).
- **miListaJugadores: static ListaJugadores** : Éste no es más que el atributo estático que contiene la única instancia de esta clase, ya que es una MAE.

MÉTODOS PRINCIPALES:

- **nuevaPartida(pTamanyoX: int, pTamanyo Y: int, pNumJugadores: int): void** : De aquí parte toda la inicialización de lógica de negocio. Los dos primeros parámetros sirven para definir el tamaño del tablero sobre el que se jugará y el tercer parámetro indica el número de jugadores que tomarán parte en la partida. Los valores de tamaño del tablero podrían ser arbitrarios, el número de jugadores ha de variar entre 2 y 4.
- **crearJugadores(pNumJugadores: int): void** : Crea tantos jugadores como se le indiquen en el parámetro de entrada y los almacena en la lista ligada circular.
- **posibleContinuarPorJugador():boolean** : Comprueba si solo queda un jugador con fichas, porque si es asi la partida deberá terminar, ya que esto significa que el resto de jugadores no tienen fichas con las que jugar.
- **obtenerGanador():Jugador** : Con ayuda del tablero recuenta las fichas que posee cada jugador y decide quien es el ganador, es decir el jugador con más fichas. Éste método será invocado cuando se llegué a una situación de bloqueo, en la cual no se puedan realizar movimientos o sea absurdo hacerlo.

3. TABLERO

DESCRIPCIÓN BREVE:

La clase Tablero dispondrá de una matriz que contenga todas las casillas necesarias para el completo desarrollo del juego. Sobre él se moverán las fichas y será capad de decidir a donde se puede mover una ficha.

Esta clase seguirá el patrón Singleton, puesto que solo deseamos un único tablero en toda la partida.

ATRIBUTOS

- **tablero: CasillaEstandar[][]** : Este atributo es una matriz de Casillas que se rellena al principio del juego y donde se colocan una ficha a cada jugador.
- **miTablero: Tablero** : Este atributo es la instancia única propia del patrón Singleton.
- **maxColum, maxFila: int** : Estos dos números son los que determinan las dimensiones máximas del tablero.

MÉTODOS PRINCIPALES

- **inicializarTablero(int pX, int pY, ListaLigadaCircular<Jugador> pJugadores): void** : Este método crea la matriz del tablero con las dimensiones que se le pasan como parámetros y luego introduce una CasillaEstandar en cada una de las posiciones de la matriz. Si implementamos los diferentes tipos de casillas o los "Power Ups" estos serian añadidos aquí aleatoriamente. Finalmente coloca una ficha de cada jugador en pJugadores introducido por parámetro en la cada esquina del tablero.
- **inicializarTablero(String pTXT): void** : Inicializa el tablero a partir de un String que contiene las posiciones de las Fichas en el tablero. Recorre el String y va analizándolo según un código, hasta completar el tablero.
- **siguienteEsquina(int numItr): Posicion** : Este método privado lo usa el anterior para obtener las diferentes esquinas donde colocar las fichas.
- **posibleContinuarPorTablero(): Boolean** : Este método analiza toda la estructura y decide si es posible realizar algún movimiento por parte de algún jugador aunque no le corresponda el turno.
- **contarFichas(Jugador pJugador): int** : Este método cuenta las fichas que hay sobre el tablero del jugador introducido. En este método se utiliza la librería JGA.
- **hayFichaAlRededor(Posicion pPos): Boolean** : Este método mira alrededor de la posición del parámetro para decidir si hay una ficha alrededor.
- **casillasLibresADistancia(int pDistancia, Posicion pPos): LinkedList<Posicion>** : Este método devuelve un iterable con todas las posiciones a distancia pDistancia de la posición indicada en el parámetro.
- **ponerFicha(Posicion pPos, Ficha pFicha): void** : Este método coloca la ficha pFicha en la posición indicada en pPos.
- **convertirAdyacentes(Posicion pPos, Jugador pNuevoDuenyo): void** : Este método hace que las fichas adyacentes a pPos cambien de jugador a pNuevoDuenyo. Si no hay ficha en una casilla o si hay una del mismo jugador no hace nada en esa casilla.
- **borrarFicha(Posicion pPos): void** : Este método elimina la ficha que contiene la posición pPos en el tablero. Si esta vacía no hace nada.

- **moverFicha(Posicion pPosFicha, Accion pAccion, Posicion pPosObjetivo, Accion pRepSel):void** : Este método selecciona del tablero la ficha que este en pPosFicha y le pasa la acción a realizar, la posición donde hacerlo y en caso de ser replicante que tipo de ficha crear.
- **calcularAccionesDe(Posicion pPos) : LinkedList<Accion>** : Devuelve las acciones posibles que puede realizar la ficha en la posición parametrizada.
- **existenCoordenadas(Posicion pPos): Boolean** :Este método decide si la posición pPos esta dentro de las posiciones posibles del tablero.
- **existeFichaEnPosicion(Posicion pPos): Boolean** : Este método decide si en la casilla de la posición pPos esta ocupada por una ficha.
- **fichaEsDeJugador(Posicion pPos, Jugador pJugador): Boolean** : Este método decide si la ficha en la posición pPos pertenece al jugador pJugador.
- **notificarObservers()**: void : Ejecuta el método notify() del patrón Observer-Observable
- **calcularPosiblesMovimientos(Posicion pPosFicha, Accion pAccSel): LinkedList<Posicion>** : Calcula las posiciones a la que la ficha en la posición pPosFicha puede acceder con la acción pAccSel.
- **getReplicPosiblesCasilla(Posicion pPos): LinkedList<Accion>** : Devuelve los tipos de replicación que puede realizar esa ficha en pPos.
- **toString(): String** : Devuelve el tablero codificado en forma de String.

4. TABLEROHEX

DESCRIPCIÓN BREVE:

La clase TableroHex es análoga a la clase Tablero. Contiene los mismos métodos, aunque con una implementación algo diferente.

Antes de seguir hablando de esta clase merece la pena recordar que no forma parte de la línea de desarrollo principal y por lo tanto no la utilizamos en nuestra aplicación, de todas formas resulta interesante conocer como se han implementado algunos aspectos de la misma.

La principal diferencia con la clase Tablero es el modo de almacenar las casillas. Al diseñar la clase tablero contábamos con casillas cuadradas, como en la mayoría de los juegos de mesa, pero tras pensar en como hacer el juego más interesante decidimos intentar crear un tablero compuesto de hexágonos. Dentro de las diferentes implementaciones para un tablero de hexágonos decidimos utilizar un grafo.

Dentro del grafo cada nodo representaría una casilla y las conexiones entre nodos representarían la adyacencia de los hexágonos.

ATRIBUTOS DE LA CLASE:

- **int [] listaVertices**: Como ya podemos deducir de la anterior explicación uno de los atributos de nuestra clase será la lista de vértices del grafo. La estructura usada es un array simple debido a la velocidad de acceso a posiciones concretas, el único modo de acceso que utilizaremos.
- **LinkedList<Integer> [] listaAdyacencia**: Esta es la otra mitad del grafo, controla las adyacencias entre nodos. La elección de un array de listas ligadas sobre una matriz de adyacencia se basa en la cantidad de memoria disponible. Cada vértice estará unido a lo sumo con otros 6, una matriz desperdiciaría cantidades inmensas de memoria.

MÉTODOS MÁS RELVANTES:

Los métodos más relevantes son los mismos que los de la clase Tablero, por lo tanto no repetiremos lo ya escrito. Los métodos difieren únicamente en la implementación ya que se ha de recorrer un grafo en vez de una matriz. Se han aplicado algunos algoritmos exclusivos de los grafos como el recorrido en anchura para hallar las casillas adyacentes a una seleccionada.

5. CASILLAESTANDAR

DESCRIPCIÓN BREVE:

Esta clase es el contenido individual del tablero. Cada una de estas casillas puede contener una entidad, que hasta ahora solo serán fichas, pero se dejo así para poder añadir mas adelantes otros elementos como los *power up*. Se puede apreciar que tiene dos constructoras, una de ellas con parámetros y la otra vacía, dependiendo de si queríamos iniciar la casilla con algún elemento o vacía.

ATRIBUTOS:

- **contenido : Entidad** : Este atributo será lo que la casilla contenga. En caso de no contener nada el valor de este atributo será *null*.

MÉTODOS PRINCIPALES:

- **CasillaEstandar()** : Esta es la constructora vacía. Crea una casilla vacía, es decir con *contenido* a *null*.
- **CasillaEstandar(Entidad pContenido)** : Esta es la constructora sobrecargada. Crea una casilla con pContenido como el contenido de esta.
- **contieneFicha():boolean** : Este método decide si la casilla contiene una ficha o no.
- **rellenar(Entidad pEntidad): void** : Este método introduce una entidad a la casilla.
- **moverFicha (Accion pAccion, Posicion pPosObjetivo, Accion pRepSel): void** : Este método mueve la ficha de esta casilla a la posObjetivo realizando la acción seleccionada.
- **calcularAccionesDe(): Void** : Devuelve la lista de acciones posible de la ficha en la casilla.
- **convertirFicha (Jugador pNuevoJ): void** : Este método cambia al dueño de la ficha que contiene, si el dueño es el mismo o esta vacía, no hace nada.
- **borrarFicha(): void** : Este método elimina el contenido de la casilla si es una ficha, no hace nada si se trata de un *power up* o esta vacía.
- **esDeJugador(Jugador pJugador): boolean** : Este método decide si la ficha que contiene es del jugador pJugador, si la casilla esta vacía o no tiene una ficha no hace nada.
- **calcularRango(Accion pAccSel): int** :devuelve el rango que la ficha contenida puede alcanzar según la Accion parametrizada.
- **getReplicPosibFicha(): LinkledList<Accion>** : Devuelve la lista de tipos de replicaciones que puede hacer esa ficha.
- **toString(): String** : Devuelve el contenido de la casilla mediante un String.

6. ENTIDAD

DESCRIPCIÓN BREVE:

Esta clase, pese a solo contener la posición en la que se encuentra en el tablero, también es el padre de todo lo que una casilla puede contener. De esta clase descenderán la clase *Ficha* y *Power up* en caso de que lleguen a desarrollarse por completo. Contendrá un getter de la posición.

ATRIBUTOS:

- **posición: Posicion** : Es la posición en la que se encuentra dicha entidad en el tablero.

MÉTODOS PRINCIPALES:

- **Entidad(Posicion pPos)** : La constructora de la entidad. Necesita una posición para poder completar la clase.

7. FICHA

DESCRIPCIÓN BREVE:

Esta clase desciende de la clase Entidad y representa una ficha en el tablero. Es una clase abstracta y nunca llegara a instanciarse, para ello, sus estarán sus "hijos", que serán las verdaderas fichas sobre el tablero, con diferentes cualidades para sus diferentes funciones.

ATRIBUTOS:

- **propietario: Jugador** : Es el dueño de la ficha y el único que puede usarla.

MÉTODOS PRINCIPALES:

- **Ficha(Jugador pJugador, Posicion pPos)** : La constructora de la ficha, necesita su dueño y la posición en la que se encuentra.
- **convertir(Jugador pJugador): void** : Este método cambia al propietario de la ficha por el nuevo parametrizado, es necesario a la hora de convertir todas las fichas adyacentes.
- **getJugador(): Jugador** : Este método devuelve el jugador al que pertenece la ficha, necesario para que sus hijos tengan acceso a él.
- **esDeJugador(Jugador pJugador): Boolean** : Este método compara al jugador introducido con el dueño de la ficha.
- **moverFicha(Accion pAccion, Posicion pPosObjetivo, Accion pRepSel): void** : Con este método, la ficha toma el control.
- **calcularAcciones(): LinkedList<Accion>** : Este método abstracto calcula las acciones que la ficha puede realizar.
- **calcularRango(Accion pAccion): int** : Este método abstracto calcula la distancia de la acción que se va a realizar, puede ser diferente para diferentes acciones.
- **realizarAccion(Accion pAccion, Posicion pObjetivo): void** : Este método abstracto realiza la acción pAccion en la posición pObjetivo.
- **getReplicPosibles()**: **LinkedList<Accion>** : Este método abstracto devuelve una lista con la capacidad que tiene de replicar la ficha.
- **toString()**: **String** : Devuelve la ficha codificada en un String incluyendo el tipo de ficha que es y el jugador al que pertenece.

8. INTERFACES ISALTADOR E IREPLICANTE

DESCRIPCIÓN:

Estas dos interfaces sirven para definir el tipo de fichas que existirán en nuestra partida. Añadiendo más interfaces podríamos crear más tipos de ficha.

Las interfaces en este caso tienen una triple funcionalidad: definir las propiedades semánticas de una clase que implemente esta interfaz, así sabemos si nuestra clase (que será además una ficha) es capaz de saltar o replicarse. Por otra parte obliga a las fichas que vayan a ser saltadores a implementar el método saltar(), o a las que vayan a ser replicantes a implementar el método replicar(). Por último contienen un atributo estático que indica la longitud del salto que pueden realizar o la distancia a la cual una ficha se puede replicar.

Con este mecanismo tenemos una forma fácil de ampliar el catálogo de funcionalidades que puedan tener las fichas, además de poder hacer que una ficha de repente obtenga una funcionalidad especial con suma facilidad.

9. SALTADORREPLICANTE

DESCRIPCIÓN BREVE:

Esta clase desciende de la clase Ficha e implementa las interfaces IReplicante e ISaltador, con lo que le dotamos a la clase con las funcionalidades de saltar y replicarse.

ATRIBUTOS:

Además de los atributos heredados de sus padres, dispone de un LinkedList<Accion> que contiene la lista de posibles tipos de replicación.

MÉTODOS PRINCIPALES:

- **SaltadorReplicante(Jugador pPropietario, Posicion pPosicion)** : Es la constructora de la ficha e invocara a la constructora de su padre.
- **getReplicPosibles()**: **LinkedList<Accion>** : devuelve la lista de posibles tipos de replicación.
- **calcularAcciones(): LinkedList<Accion>** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Devuelve la lista de acciones que puede hacer la ficha, en este caso SALTAR y REPLICAR.
- **calcularRango(Accion pAccion): int** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Devuelve un número que es la distancia en la que puede ejecutar la acción introducida.
- **realizarAccion(Accion pAccion, Posicion pObjetivo): void** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Según la acción llamara a los métodos implementados de las interfaces que correspondan.
- **saltar(Posicion pObjetivo): void** : Este método es la implementación de su correspondiente en la interfaz ISaltador. Mueve la ficha de la actual a la casilla indicada.
- **replicar(Posicion pObjetivo): void** : Este método es la implementación de su correspondiente en la interfaz IReplicante. Crea una nueva ficha del tipo seleccionado en la casilla indicada.
- **toString(): String** : Devuelve un string con el código de su tipo de ficha y el numero de jugador que le posee.

10. SALTADOR

DESCRIPCIÓN BREVE:

Esta clase desciende de la clase Ficha e implementa la interfaz ISaltador, con lo que le dotamos a la clase con la funcionalidad propia de este tipo de ficha.

ATRIBUTOS:

No dispone de atributos más allá de los heredados de sus padres.

MÉTODOS PRINCIPALES:

- **Saltador (Jugador pPropietario, Posicion pPosicion)** : Es la constructora de la ficha e invocara a la constructora de su padre.
- **calcularAcciones(): LinkedList<Accion>** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Devuelve la lista de acciones que puede hacer la ficha, en este caso SALTAR y REPLICAR.
- **calcularRango(Accion pAccion): int** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Devuelve un número que es la distancia en la que puede ejecutar la acción introducida.
- **realizarAccion(Accion pAccion, Posicion pObjetivo): void** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Según la acción llamara a los métodos implementados de las interfaces que correspondan.
- **saltar(Posicion pObjetivo): void** : Este método es la implementación de su correspondiente en la interfaz ISaltador. Mueve la ficha de la actual a la casilla indicada.
- **getReplicPosibles()**: LinkedList<Accion> : devuelve un null, debido a que un saltador no puede replicar ningún tipo de ficha.
- **toString(): String** : Devuelve un string con el código de su tipo de ficha y el numero de jugador que le posee.

11. REPLICANTE

DESCRIPCIÓN BREVE:

Esta clase desciende de la clase Ficha e implementa la interfaz IReplicante, con lo que le dotamos a la ficha con esa funcionalidad.

ATRIBUTOS:

Además de los atributos heredados de sus padres, posee un LinkedList<Accion> que posee los tipos de ficha que puede replicar.

MÉTODOS PRINCIPALES:

- **SaltadorReplicante(Jugador pPropietario, Posicion pPosicion)** : Es la constructora de la ficha e invocara a la constructora de su padre.
- **calcularAcciones(): LinkedList<Accion>** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Devuelve la lista de acciones que puede hacer la ficha, en este caso SALTAR y REPLICAR.
- **calcularRango(Accion pAccion): int** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Devuelve un número que es la distancia en la que puede ejecutar la acción introducida.
- **realizarAccion(Accion pAccion, Posicion pObjetivo): void** : Este método es la implementación del método con el mismo nombre que se encontraba en la clase padre. Según la acción llamara a los métodos implementados de las interfaces que correspondan.
- **replicar(Posicion pObjetivo): void** : Este método es la implementación de su correspondiente en la interfaz IReplicante. Crea una nueva ficha del tipo seleccionado en la casilla indicada.
- **toString(): String** : Devuelve un string con el código de su tipo de ficha y el numero de jugador que le posee.
- **getReplicPosibles()**: **LinkedList<Accion>** : devuelve la lista de posibles tipos de replicación.

12. GESTORGUARDADO

DESCRIPCIÓN BREVE

Esta clase se encarga de guardar y cargar datos de unos ficheros para poder dotar al juego con capacidad de auto-guardado. A través de los métodos toString() de las diferentes clases de la lógica de negocio recolecta información codificada sobre ellas y los une en una cadena de texto que es guardada en un archivo.

Aunque hemos hecho pruebas y utilizado esta clase exitosamente durante las pruebas con la interfaz por consola no hemos tenido tiempo para adaptarla a la nueva versión con interfaz gráfica, por lo que no se usa en la versión que le entregamos. Funcionaba como tal, pero daba lugar a unos cuantos problemas inesperados.

GestorGuardado es además una MAE, ya que comprende un subsistema independiente del resto en la lógica de negocio y debe ser único. Ha sido muy útil para realizar pruebas, al poder crear cualquier escenario y cargarlo para ponerlo en marcha sobre el tablero, además de dotar al juego de una funcionalidad muy útil.

ATRIBUTOS

- **miGestorGuardado: GestorGuardado** : Atributo para el patrón Singleton (MAE)
- **NOMBRE_FICHERO_GUARDADO: final String y NOMBRE_FICHERO_CARGA: final String** : Ambos atributos guardan el nombre de los ficheros de guardado y carga respectivamente (que normalmente serán el mismo).

MÉTODO PRINCIPALES

- **guardarPartida():void** : Recolecta información de las clases de la lógica de negocio, la une y la graba en un fichero de texto.
- **cargarPartida():void** : Toma la información de un fichero de texto, la divide e inicializa las MAEs de la lógica de negocio con la información extraída. De esta forma podemos comenzar una partida en una situación arbitraria.

INTERFAZ

INTRODUCCIÓN

En un comienzo, se realizo una interfaz por consola, muy simple e intuitiva. Funcionaba mediante Scans y prints. Nos sirvió para comprobar que la lógica de negocio funcionaba a la perfección y sin ningún problema.

En clase aprendimos la librería Swing y decidimos cambiar esta minimalista interfaz por consola por una realizada en Swing. Esto nos trajo numerosos inconvenientes, como el cambiar una programación secuencia, que era lo que se había desarrollado, a una programación dirigida por eventos. Esto nos obligo a cambiar muchas cosas, sobretodo métodos que interaccionaban con el usuario final. Además incluimos eventos en nuestro diseño.

Logramos crear mediante el uso de varias ventanas, una interfaz que nos ofreciese una unión entre la lógica de negocio y los usuarios finales.

A continuación se encuentran detalladas las principales clases de las interfaces que hemos implementado.

1. INICIOJUGAR

BREVE DESCRIPCIÓN

Esta clase es la que inicia la partida de nuestro juego. Extiende de la clase JFrame y esta capacitada para ofrecer al usuario si quiere iniciar una nueva partida o cargar una existente mediante dos botones. Ademas dispone de la información de los autores en un JLabel. Actualmente no se puede usar la funcionalidad del cargado, pese a que en nuestro proyecto esta dotado para poder hacerlo.

ATRIBUTOS DE LA CLASE

- **contentPanel: JPanel** : Es el panel que contiene los JButton y el JLabel anteriormente citado.

MÉTODOS MÁS RELEVANTES

- **main(): void** : Es el método estático que nos permite ejecutar la aplicación. Solamente lanza esta ventana.
- **InicioJugar(): void** : Es la constructora de la clase, donde se crean y colocan los botones y etiquetas y se localizan los dos eventos de los botones.
- **cerrarVentana(): void** : Este método esta implementado en todas las ventanas y simplemente cierra la ventana y la elimina.

2. VENTANANUEVAPARTIDA

BREVE DESCRIPCIÓN

Esta nueva ventana permite, mediante dos comboBox seleccionar el tamaño del tablero que se va a jugar y el número de jugadores que van a jugar. Así mismo contiene otros dos botones que permiten continuar la aplicación con los datos seleccionados en las comboBox y otro que cierre la aplicación.

ATRIBUTOS DE LA CLASE

- **contentPanel: JPanel** : Es el panel donde esta contenido el resto de los elementos de la ventana.
- **numJug: int** : Es el resultado seleccionado de la comboBox que contiene el numero de jugadores.
- **comboBox: JComboBox** : Es la comboBox que contiene el tamaño del tablero y tal y como esta diseñado el juego, actualmente solo dispone de los valores 10, 20 y 30.
- **comboBox2: JComboBox** : Es la comboBox que contiene el numero de jugadores que disputaran la partida, hasta ahora puede ser disputado desde 2 hasta 4 jugadores.

MÉTODOS MÁS RELEVANTES

- **VentanaNuevaPartida(): void** : Es la constructora de la ventana y coloca en ella todos los componentes necesarios. También asigna los eventos a los botones de aceptar y cancelar.

3. VENTANAJUEGO

BREVE DESCRIPCIÓN

Esta ventana es la representación gráfica del tablero. Esta dividido en tres partes, la primera contiene puntuaciones de dos de los jugadores, luego está la más amplia que contiene los botones para realizar la partida y por último hay otra que contiene más puntuaciones de los jugadores.

Las puntuaciones están realizadas mediante labels que indican el jugador y la puntuación obtenida en número de fichas que dispone. Cada jugador dispone de un color diferente.

El tablero en sí esta representado por una matriz de botones. Todos ellos tienen el mismo evento que se describirá en la tabla de eventos.

ATRIBUTOS DE LA CLASE

- **tamTablero: int** : guarda el tamaño del tablero, es un único número ya que el tablero es cuadrado, pese a que estaba diseñado para que no fuera necesariamente así.
- **jugadorActual: Jugador** : guarda el jugador al que le toca el turno, pudiendo de esta manera, avisar de a quien pertenece el turno, además de permitir la realización de comprobaciones sobre las reglas de juego.
- **Diversos JPanel** : Sirven para almacenar los diferentes componentes en los lugares adecuados.
- **Diversos JLabel** : Para guardar el nombre de los jugadores y su puntuación.
- **Botones: JButton[]** : Un array de botones que nos permite el acceso a los botones sin tener que buscarlos por los paneles.
- **casillaEvent: CasillaPinchada** : Este atributo es una clase que extiende MouseAdapter. Se trata de un recogedor de eventos común a todos los botones del tablero visual, por ello se ha creado como clase aparte.

MÉTODOS MÁS RELEVANTES

- **VentanaJuego(pTamanyo: int, pNumJug: int) : void** : Es la constructora de la ventana, recibe como parámetros los datos del tamaño y el numero de jugadores. Con ellos llama a inicarVentana() para que rellene todos los campos de manera correcta.
- **iniciarVentana(): void** : En este método se rellena el tablero por zonas. En el oeste se colocan los datos de dos de los jugadores y el este los otros dos jugadores, sean o no jugadores que intervienen en la partida. Tanto en el norte como en el sur se colocan paneles vacios para que quede simétrico. En el centro se colocan los botones con otro método.
- **colocarBotones(): void** : Mediante dos For coloca los botones que representan las casilla del tablero de la lógica de negocio, con unas labels indicando el número de fila y columna. Estos botones son genéricos y no tienen ninguna representación gráfica sobre ellos, pero si que contienen su evento asociado.
- **refrescar(): void** : Este método se llama en cada final de turno y mediante el String que obtiene del tablero de la lógica de negocio, imprime en cada botón el tipo de ficha contenida en cada casilla y el jugador al que pertenece. También hace un recuento de estas, y actualiza las puntuaciones de jugadores en las labels correspondientes.
- **anunciarJugador(pNumJugador:int): void** : Muestra un pequeño dialogo que avisa del turno.

- **siguienteJugador(): void** : Comprueba si se puede continuar la partida, tanto si el jugador es capad de ello, como si el tablero no esta completo. En caso de no ser asi, muestra un mensaje por pantalla del ganador y cierra la aplicación.
- **Update(): void** : método del patrón observer-observator, que refresca los botones y comprueba si se puede continuar.

4. SELECCIONARACCION, SELECCIONARREPLICACION, SELECCIONARPOSICION

BREVE DESCRIPCIÓN

Estas tres clases muy parecidas entre sí se encargan de alimentar a la maquinaria que conforma la lógica de negocio. Al hacer el jugador clic en una casilla deberá seleccionar una serie de opciones de cada una de estas 3 ventanas, más adelante los datos recolectados son transmitidos a la lógica de negocio para que ésta mueva una ficha.

ATRIBUTOS DE LAS CLASES

Ninguna de estas clases posee ningún atributo relevante. Todos los atributos que poseen no son más que objetos visuales para formar las ventanas.

MÉTODOS MÁS RELEVANTES

No se dan métodos relevantes como tal, ya que estas clases son muy sencillas y se componen de un solo MouseAdapter que reconoce los clics sobre el botón ”OK”. Dentro de estos MouseAdapters se suelen ejecutar una serie de consultas a la lógica de negocio para planear la siguiente pregunta para el usuario y se produce una delegación del flujo del programa a la siguiente ventana.

La última ventana SeleccionarPosicion tiene algo más de miga y su evento será descrito en la tabla de eventos de la siguiente sección.

EVENTOS

En la interfaz gráfica la programación es dirigida por eventos, en lugar de ser una programación lineal. Este nuevo enfoque nos lleva a usar estos eventos, programarlos y usarlos correctamente. Para esto, y a modo de resumen, rellenaremos la siguiente tabla con los eventos que hemos manejado y creado para el correcto uso de la aplicación.

VENTANA	COMPONENTE	CLASE MANEJADOR	MÉTODO MODELO
InicioJuego	Botón NuevaPartida	`Mouseadapter`	• Llama a la VentanaNuevaPartida
InicioJuego	Botón CargarPartida	`Mouseadapter`	• No esta operativo
VentanaNuevaPartida	Botón OK	`Mouseadapter`	• Llamada a nuevaPartida(...) en ListaJugadores • Obtiene los valores de las comboBox y se las pasa a VentanaJuego como parámetros.
VentanaNuevaPartida	Botón Cancelar	`Mouseadapter`	• Cierra la aplicación.
VentanaJuego	JButton	`CasillaPinchada`	• fichaEsDeJugador(...) en Tablero • calcularAccionesDe(...) en Tablero • Lanza una ventana SeleccionarAccion
SeleccionarAccion	Botón OK	`Mouseadapter`	• Lee acción seleccionada por usuario • calcularPosiblesMovimientos(...) en Tablero • Lanza SeleccionarReplicacion o SeleccionarPosicion
SeleccionarReplicacion	Botón OK	`Mouseadapter`	• Lee el tipo de replicación seleccionado por el usuario • Lanza SeleccionarPosicion
SeleccionarPosicion	Botón OK	`Mouseadapter`	• Lee la posición elegida por el usuario • moverficha(...) de Tablero (*)

(*) moverFicha(...) desencadena una serie de acciones muy importantes sobre la lógica de negocio, de forma que las explicaremos aquí:

Éste método se encarga de hacer que las opciones seleccionadas por el usuario mediante la interfaz gráfica cobren vida en la parte de la lógica de negocio. A través de moverFicha(...) se transmite a la lógica de negocio la siguiente información:

- Que ficha ha de moverse
- Adonde ha de moverse
- Que tipo de acción debe realizar
- Y en caso de que la acción sea “replicar” se indica también que tipo de ficha se crea tras la replicación (ya que al replicarse una ficha puede crear diferentes tipos de ficha)

Toda esta información va filtrándose por la lógica de negocio hasta llegar al método realizarAccion() de un ficha concreta, el cual grabará estos cambios en el tablero de la lógica de negocio y ordenará al tablero a notificar a sus observadores para así representar los cambios sobre la parte visual.

ESTRUCTURAS

A lo largo del desarrollo de nuestra aplicación hemos tenido que usar ciertas estructuras de datos para almacenar objetos. En los casos posibles hemos utilizado las estructuras ya implementadas en las librerías de Java, pero se ha dado casos en las que necesitábamos contar con algo más personalizado, creado a medida.

Una de estas estructuras es la clase Posicion, esta clase nos ayuda a almacenar dos coordenadas en una misma estructura. La necesidad de esta clase es bien clara, ya que utilizamos un tablero de dos dimensiones y necesitamos situar las fichas en él. Internamente no cuenta con más que dos atributos int y sus respectivos getters.

La otra estructura usada ya resulta más complicada, ya que está formada por 3 clases: IListaLigadaCircular<T>, ListaLigadaCircular<T> y Nodo<T>. Como podemos imaginar si ya hemos leído el apartado sobre la clase ListaJugadores, esta estructura será la que entonces mencionábamos. Comentemos la funcionalidad de cada una de las clases individualmente:

- **IListaLigadaCircular<T>**: Esta es la interfaz que define los métodos que deberá implementar cualquier lista ligada circular.
- **ListaLigadaCircular<T>**: Esta clase es la estructura de datos en sí. Implementa la interfaz anterior. Contiene un puntero a un nodo del cual colgarán el resto de nodos de la lista. Además el último nodo siempre apuntará al primero, creando así un circulo. Para saber que nodo es el "actual" mientras recorremos la lista creamos un segundo puntero que servirá para guardar esta información.
- **Nodo<T>**: Los "ladrillos" que forman la anterior estructura. Cada nodo guarda unos datos y un puntero al siguiente nodo de la lista.

Como podemos observar las tres clases son de tipo genérico de manera que podríamos almacenar en ellas cualquier tipo de datos, aunque nosotros solo las utilizaremos para almacenar objetos de tipo **Jugador**.

EXCEPCIONES

En algunos de nuestros métodos hemos decidido lanzar excepciones propias para señalar algunos errores específicos de nuestro juego, como la selección de una posición inválida en el tablero, por ejemplo la posición (-34, 1234). Aunque la interfaz gráfica limita fuertemente los comportamientos incorrectos del usuario hemos decidido dejarlas en el programa por si se da algún caso muy especial, o algún otro programador toma el código para seguir su desarrollo y no utiliza la interfaz gráfica.

Para esta y otras comprobaciones hemos creado nuestras propias excepciones que lógicamente heredan su funcionalidad de la clase Exception de Java.

Nuestras excepciones son:

- **CoordenadasInexistentesException**: Se lanza cuando las coordenadas indicadas no existen en el tablero.
- **FichaDeOtroPropietarioException**: Utilizada cuando un jugador pretende seleccionar una ficha de la cual no es dueño.
- **FichaInexistenteException**: Esta excepción es lanzada cuando un jugador pretende seleccionar una ficha y para ello selecciona una casilla que no contenga una ficha (ya sea porque esté vacía o contenga otro tipo de entidad).
- **NumJugadoresIncorrecto**: Al crearse una nueva partida si el número de jugadores no está dentro del límite establecido esta excepción nos llamará la atención.

LIBRERÍAS Y PATRONES

A lo largo de la asignatura hemos adquirido conocimientos sobre el desarrollo del software que facilitan nuestro trabajo como ingenieros y programadores. Para aprovecharnos de dichas ventajas y a demás poner en práctica los conocimientos teóricos hemos luchado por introducir en nuestro proyecto todos aquellos conceptos, patrones y librerías que hemos visto en clase.

Por difícil que haya resultado lo hemos conseguido y aquí se plantea una lista de la aplicación de los conocimientos teóricos en nuestro proyecto:

- Diseño orientado a objetos: Aplicado sobre toda la estructura del programa. Además éste resulta ser un paradigma muy acertado para nuestro programa, ya que resulta ser un fiel reflejo de la propia realidad. Por ejemplo el tablero que contiene sus casillas. Y sobre cada casilla está cada ficha, que además de ser ficha puede ser un tipo de ficha especial.
- Modelado UML: Altamente útil en la segunda fase del proyecto (diseño). Primero pensamos como crear el programa en base a diagramas, resultando esto en una implementación muy sencilla y sin sustos.
- Tipos enumerados: El enumerado “Accion” que indica que tipos de movimientos pueden realizar nuestras fichas.
- Patrón Singleton: Utilizado en las MAEs de nuestro programa, que han sido Tablero, ListaJugadores y GestorGuardado.
- Patrón Iterator: Ya que hemos creado una estructura personalizada para nuestro proyecto también ha surgido la necesidad de recorrerla. Para ello hemos creado nuestro propio iterador “IteratorListaLigadaCircular” en base al patrón visto en clase.
- Aplicación de tres capas: Por una parte contamos con la interfaz, por otra con la lógica de negocio y por último contamos con almacenamiento de datos durante la partida para guardar el progreso.
- Herencia: Algunas de nuestras clases, un claro ejemplo son nuestras excepciones, heredan comportamientos de las clases existentes en las APIs de Java. Pero además nos hemos cruzado con el mismísimo problema del diamante en nuestro diseño a la hora de definir las fichas. Éste problema lo hemos resuelto con el uso de interfaces (ISaltador e IReplicante).
- Librería JGA: Ha sido utilizada para realizar el recuento de fichas para un jugador específico en el tablero. Para ello hemos tenido que crear nuestro propio UnaryFunctor.
- Patrón SimpleFactory: Utilizado en la interfaz gráfica para crear todos los botones del tablero que representan sus casillas. En base a un bucle creamos 100, 200 o 300 botones, todos con las mismas cualidades salvo el nombre.
- Interfaces gráficas: Hemos añadido una interfaz gráfica dirigida por eventos que se puede observar al ejecutar el programa, o consultando código.
- Manejadores compartidos: Hemos creado un mismo manejador de eventos para todos los botones del tablero, para ello hemos creado nuestro propio manejador como clase aparte que extiende MouseAdapter.
- Modelo-Vista-Controlador, patrón Observer-Observable: Implementado para beneficiarnos de sus ventajas. La lógica de negocio únicamente lanza notifyObserver()’s a la interfaz gráfica y ésta reacciona en consecuencia. Además es la interfaz la que recolecta los datos con la que luego alimenta al modelo.

CONCLUSIONES

Este proyecto nos ha servido para afianzar los conocimientos aprendidos a lo largo de la asignatura, por ello hemos tratado de incorporar todo lo que hemos visto a lo largo del cuatrimestre, como la librería genérica JGA, patrones de diseño, como Singleton, herencia e interfaces.

También agregamos la interfaz gráfica realizada con la librería Swing, que nos conllevo serios replanteamientos de la metodología de la programación que llevábamos y tuvimos que cambiar de un modelo secuencial a uno dirigido por eventos.

Nos dimos cuenta que es imposible tener un diseño inicial y ajustarse a este al pie de la letra, puesto que salen inconvenientes, o necesidades no previstas que nos obligan a añadir, modificar o quitar cosas que en un principio no se consideraban necesarios.

Como conclusión de éste proyecto pensamos que los patrones y conceptos vistos en clase son útiles, pero únicamente si se adaptan a un proyecto que se puede beneficiar de ellos. Si se intentan aplicar “con calzador” funcionarán, pero quizás el esfuerzo extra que ha sido invertido en implementarlos no haya merecido la pena. Cada proyecto es un mundo y tiene sus propias necesidades que deben ser atendidas adecuadamente y no en base a unos requerimientos externos.