Práctica 2:

Implementación de la estructura de clases de un sistema orientado a objetos, dado su diseño estructural

Desarrollo de esta práctica

En esta práctica se realizará la implementación del diagrama de clases completo de Irrgarten. Este diagrama se suministra adjunto a este guion. En concreto se crearán todas las clases, para cada clase se definirán sus atributos, incluyendo aquellos que surgen de las relaciones entre clases, se definirán las cabeceras de todos los métodos, y se implementarán algunos de ellos. Los métodos que queden sin implementar en esta práctica, y que se indica en este guion cuáles son, se implementarán en la práctica siguiente.

Al inspeccionar el citado diagrama de clases te darás cuenta que algunas de clases ya las implementaste en la práctica anterior. Efectivamente es así y esa parte del trabajo ya la tienes completada.

Algunos de los métodos que implementarás necesitan realizar llamadas a otros métodos cuya especificación recibirás en el siguiente guion. Esto no debe suponer un problema para el desarrollo de esta práctica ya que puedes hacer llamadas a métodos de los cuales solo dispongas de su cabecera.

Un objetivo importante de esta práctica es continuar con tu aprendizaje de diseño de software, por ello es fundamental que no te quedes solamente en los detalles de implementación o en las particularidades de los lenguajes; debes analizar el diagrama de clases y observar cómo este diseño representa el juego que se está desarrollando y cómo las distintas clases tienen unas responsabilidades muy concretas siguiendo los principios de alta cohesión y bajo acoplamiento.

Añadiendo nuevas clases

Además de las clases ya suministradas, deberás implementar las siguientes:

- Monster
- Player
- Labyrinth
- Game

Debes añadir todos los atributos y la cabecera de todos los métodos del diagrama (deja el cuerpo de los métodos nuevos vacío). Presta atención a la visibilidad de cada elemento. En Java, para evitar problemas de compilación, puedes utilizar la siguiente sentencia como única línea del cuerpo de los métodos de los que por ahora solo tienes la cabecera:

```
throw new UnsupportedOperationException();
```

Debes además añadir todos los atributos generados por las relaciones entre las distintas clases. Todos esos atributos producidos por asociaciones serán privados.

Añadiendo funcionalidad

En este apartado crearás el cuerpo de una serie de métodos sencillos para ir completando la funcionalidad del juego. Los métodos más complejos se abordarán en la práctica siguiente.

Además de los métodos indicados a continuación debes añadir los constructores de las nuevas clases indicados en el diagrama.

Monster

boolean dead(). Devuelve *true* si el monstruo está muerto.

float attack(). Genera el resultado delegando en el método *intensity* del dado pasando como parámetro la fuerza del monstruo.

void setPos(int row, int col). Es un modificador en una única llamada de los atributos *row* y *col*.

String toString(). Genera un representación del estado completo del monstruo en forma de cadena de caracteres.

void gotWounded(). Este método decrementa en una unidad el atributo que representa la salud del monstruo.

boolean defend(float receivedAttack). De este método recibirás información en la siguiente práctica.

Player

Por simplicidad, en Java, los contenedores que se utilizarán serán del tipo *ArrayList* y en Ruby de tipo *Array*.

En el constructor de esta clase se debe inicializar de oficio el atributo nombre. El nombre de los jugadores se establece concatenando la cadena “Player #” y su número.

void resurrect(). Este método realiza las tareas asociadas a la resurrección. Debe hacer que las listas de armas y escudos sean listas vacías, que el nivel de salud sea el determinado para el inicio del juego y el número consecutivo de impactos cero.

void setPos(int row, int col). Es un modificador en una única llamada de los atributos *row* y *col*.

boolean dead(). Devuelve *true* si el jugador está muerto.

f*loat attack()*. Calcula la suma de la fuerza del jugador y la suma de lo aportado por sus armas (*sumWeapons*).

boolean defend(float *receivedAttack*). Este método delega su funcionalidad en el método *manageHit*.

String toString(). Genera un representación del estado completo del jugador en forma de cadena de caracteres.

Weapon newWeapon(). Genera una nueva instancia de arma. Los parámetros necesarios para construir el arma se le solicitarán al dado.

Shield newShield(). Genera una nueva instancia de escudo. Los parámetros necesarios para construir el escudo se le solicitarán al dado.

f*loat defensiveEnergy()*. Calcula la suma de la inteligencia con el aporte de los escudos (*sumShields*).

void resetHits(). Fija el valor del contador de impactos consecutivos a cero.

void gotWounded(). Este método decrementa en una unidad el atributo que representa la salud del jugador.

void incConsecutiveHits(). Incrementa en una unidad el contador de impactos consecutivos.

float sumWeapons(). Devuelve la suma del resultado de llamar al método *attack* de todas sus armas.

float sumShields(). Devuelve la suma del resultado de llamar al método *protect* de todos sus escudos.

De los siguientes métodos:

- *Directions move(Directions direction,ArrayList<Directions> validMoves)*
- *void receiveReward()*
- *void receiveWeapon(Weapon w)*
- *void receiveShield(Shield s)*
- *boolean manageHit(float receivedAttack)*

recibirás información en la siguiente práctica.

Labyrinth

En esta clase se ha decidido que la relación entre la misma y los objetos *Square se represente mediante tablas, quedando así la información relativa a filas y columnas almacenada de forma implícita. Es una forma menos óptima en cuanto a requisitos de almacenamiento para representar un tablero de elementos del laberinto, monstruos y jugadores (sobre todo si la mayor parte está vacía) pero más eficiente en cuanto al acceso y simple a nivel conceptual.

De esta forma, el laberinto tendrá una tabla de dimensiones *nRows* X *nCols* que permitirá almacenar referencias a monstruos. Equivalentemente tendrá otra tabla de igual tamaño para poder almacenar referencias a jugadores. En tercer lugar, dispondrá de otra tabla más que permitirá almacenar

caracteres que representen el estado de cada casilla del laberinto (‘X’=obstáculo, ‘-’ para representar una casilla vacía, ‘M’=monstruo, ‘E’=casilla de salida y ‘C’=casilla que contiene a la vez un monstruo y un jugador y donde potencialmente se está produciendo un combate. Todas estás tablas se implementarán usando *arrays* estáticos bidimensionales en Java (Monster[ ][ ], Player[ ][ ] y char[ ][ ]) y *Arrays* de *Arrays* en Ruby (un *Array* que representa las filas donde cada elemento es a su vez otro *Array* que permita almacenar los elementos de todas las columnas de esa fila).

Recuerda que la **casilla de salida** es la que contiene la puerta para abandonar el laberinto y ganar la partida y que los jugadores son situados al azar dentro del laberinto al inicio del juego.

Cuando diseñes tu laberinto, este será, por tanto, una matriz bidimensional de celdas cuadradas, donde los muros del laberinto también ocupan una celda completa, etiquetada como ‘X’ (obstáculo). La primera coordenada referencia la fila y la segunda coordenada referencia la columna. Así, un laberinto de 5x7 será de 5 celdas de alto (5 filas) y 7 celdas de ancho (cada fila tendrá 7 celdas). Los índices, considéralos de forma que la celda (0,0) es la que se encuentra en la esquina superior izquierda. Todos los métodos de las prácticas que implican *moverse* por el laberinto están especificados en los guiones teniendo en cuenta que la celda (0,0) es la que se encuentra en esa posición concreta.

Seréis vosotros los que diseñaréis vuestro laberinto, indicando sus dimensiones, situando la casilla de salida y los obstáculos. Sobre todo al principio, experimentad con laberintos de pequeño tamaño que os permitan probar distintos tipos de situaciones.

Por simplicidad, en Java, los contenedores que se utilizarán serán del tipo *ArrayList* salvo en el caso en que se presente una pareja fila-columna donde se utilizarán *arrays* estáticos de dos datos tipo int (métodos *dis2pos* y *randomEmptyPos)*. En Ruby todos los contenedores usados serán del tipo Array.

boolean haveAWinner(). Devuelve true si hay un jugador en la casilla de salida y false si no hay ninguno.

String toString(). Genera un representación del estado completo del laberinto en forma de cadena de caracteres.

addMonster(int row,int col, Monster monster). Si la posición suministrada está dentro del tablero y está vacía, anota en el laberinto la presencia de un monstruo, guarda la referencia del monstruo en el atributo contenedor adecuado e indica al monstruo cual es su posición actual (*setPos*).

boolean posOK(int row, int col). Devuelve *true* si la posición proporcionada está dentro del laberinto.

emptyPos(int row, int col). Devuelve *true* si la posición suministrada está vacía.

monsterPos(int row, int col). Devuelve *true* si la posición suministrada alberga exclusivamente un monstruo.

exitPos(int row, int col). Devuelve *true* si la posición suministrada es la de salida.

combatPos(int row, int col). Devuelve *true* si la posición suministrada contiene a la vez un monstruo y un jugador (carácter ‘C’).

boolean canStepOn(int row, int col). Indica si la posición suministrada está dentro del laberinto y se corresponde con una de estas tres opciones: casilla vacía, casilla donde habita un monstruo o salida (*posOK, emptyPos,* monsterPos, *exitPos*).

updadeOldPos(int row, int col). Este método solo realiza su función si la posición suministrada está dentro del laberinto. Si es el caso, si en esa posición el laberinto estaba indicando el estado de combate, el estado de esa casilla del laberinto pasa a indicar que simplemente hay un monstruo. En otro caso, el estado de esa casilla del laberinto pasa a indicar que está vacía. Este método es llamado cuando un jugador abandona una casilla y se encarga de dejar la casilla que se abandona en el estado correcto.

i*nt[] dir2Pos(int row, int col, Directions direction)*. Este método calcula la posición del laberinto a la que se llegaría si desde la posición suministrada se avanza en la dirección pasada como parámetro. No es necesario realizar comprobaciones relativas a no generar posiciones fuera del laberinto.

int[] randomEmptyPos(). Utilizando el dado, genera una posición aleatoria en el laberinto (fila y columna) asegurando que esta esté vacía. Genera internamente posiciones hasta que se cumple esta restricción y una vez generada se devuelve. Si no hay posiciones vacías se producirá un bucle infinito.

De los siguientes métodos:

- *spreadPlayers(ArrayList<Player> players).*
- *Monster putPlayer(Directions direction, Player player)*
- *void addBlock(Orientation orientation, int startRow,int startCol, int length)*
- *ArrayList<Directions> validMoves(int row,int col)*
- *Monster putPlayer2D*

recibirás información en la siguiente práctica.

Game

En el constructor hay que crear los jugadores y añadirlos al contenedor adecuado, determinar quien va a empezar y fijar el jugador con el turno. Inicializa también el atributo contenedor de monstruos e instancia un laberinto para inicializar el atributo *labyrinth*. Inicializa el resto de atributos con los valores iniciales que consideres apropiados. Para finalizar, se debe llamar al método que configura el laberinto y al que reparte los jugadores por el mismo.

Por simplicidad, en Java, los contenedores que se utilizarán serán del tipo *ArrayList* y en Ruby de tipo *Array*.

boolean finished(). Delega en el método del laberinto que indica si hay un ganador.

GameState getGameState(). Genera una instancia de *GameState* integrando toda la información del estado del juego.

void configureLabyrinth(). Configura el laberinto añadiendo bloques de obstáculos y monstruos. Los monstruos, además de en el laberinto son guardados en el contenedor propio de esta clase para este tipo de objetos.

void nextPlayer(). Actualiza los dos atributos que indican el jugador (current*) con el turno pasando al siguiente jugador.

void logPlayerWon(). Añade al final del atributo *log* (concatena cadena al final) el mensaje indicando que el jugador ha ganado el combate. También añade el indicador de nueva línea al final.

void logMonsterWon(). Añade al final del atributo *log* (concatena cadena al final) el mensaje indicando que el monstruo ha ganado el combate. También añade el indicador de nueva línea al final.

void logResurrected(). Añade al final del atributo *log* (concatena cadena al final) el mensaje indicando que el jugador ha resucitado. También añade el indicador de nueva línea al final.

void logPlayerSkipTurn(). Añade al final del atributo *log* (concatena cadena al final) el mensaje indicando que el jugador ha perdido el turno por estar muerto. También añade el indicador de nueva línea al final.

void logPlayerNoOrders(). Añade al final del atributo *log* (concatena cadena al final) el mensaje indicando que el jugador no ha seguido las instrucciones del jugador humano (no fue posible). También añade el indicador de nueva línea al final.

void logNoMonster(). Añade al final del atributo *log* (concatena cadena al final) el mensaje indicando que el jugador se ha movido a una celda vacía o no le ha sido posible moverse. También añade el indicador de nueva línea al final.

void logRounds(int rounds,int max). Añade al final del atributo *log* (concatena cadena al final) el mensaje que se han producido rounds de max rondas de combate. También añade el indicador de nueva línea al final.

De los siguientes métodos:

- *boolean nextStep(Directions preferredDirection)*
- *actualDirection(Directions preferredDirection)*
- *GameCharacter combat(Monster monster)*
- *manageReward(GameCharacter winner)*
- *void manageResurrection()*

recibirás información en la siguiente práctica.