Sobre los Efectos de Entidad

Entidad tiene un método abstracto afectar(), el cual es redefinido por cada una de las clases concisas que lo implementan. afectar() modela lo que sucede cuando Pacman interactua con una entidad (el afectar() de pacman no hace nada).

Para poder usar el afectar() de manera apropiada, las Entidades conocen a la grilla que las contiene (para poder realizarle preguntas y pedidos directamente, así como para poder comunicarse con la clase Juego usando un Mediator si fuera necesario)

En simultáneo, Grlla conoce a algunas entidades precisas, como serían Pacman y los Fantasmas para poder afectarlos.

Ejemplos de Efectos de Entidad:

Moneda:

```
public void afectar() {
        grilla.pedirSumarPuntos(200)
        grilla.seColeccionoUnaMoneda()
}
```

Power Pellet:

```
public void afectar() {
        grilla.ponerFantasmasEnRun()
}
```

Fantasma:

```
public void afectar() {
        miEstado.afectar()
}
```

Sobre renderizar Entidades:

Bloque tiene un campo boolean "esPared", para determinar si se puede mover na EM a ese casillero (naturalmente, si es pared, NO se pueden mover). Luego, el método de mostrar imagen se vería algo como:

```
-------------------------------------------------------------------------

if(esPared)

        return caminoImagenPared;

else if(iteradorDeEntidades.isEmpty())

        return caminoImagenPiso;
else

        return caminoMaximaPrioridad();

-------------------------------------------------------------------------
```

Donde caminoMaximaPrioridad() es un método privado que recorre el iterador y retorna el caminoimagen de la Entidad con mas prioridad en él.

Sobre el Movimiento de Pacman

Cada grupo de Entidades Móviles (O sea, Pacman, y los Fantasmas) tienen un reloj "propio" asociado a grilla. Este reloj le pide, cada un determinado intervalo, mover a esas entidades.

Grilla le pide a las EM la posición a la que se van a poder mover. Las EM calculan esta posición con su metodo getPosicionSiguiente(). En ese mismo proceso, le preguntan a Grilla si la posición a la que se desean mover es una pared, o bien si no existe, por ejemplo (-1, 0). Si no se da ninguno de esos casos, las EM devuelven la posición deseada. Sino, devuelven su posición actual.

Grilla luego efectúa el movimiento como tal, removiendo a la EM del iterador del bloque en el que están, y agregándola al bloque correspondiente a la siguiente posición.

Pacman define su siguiente movimiento como 1 bloque más adelante, donde adelante es la dirección en la que está apuntando.

La GUI captará los eventos del keyPress y podrá cambiar la dirección:

Arriba -> NORTE

Abajo -> SUR

Derecha -> ESTE

Izquierda -> OESTE

Una vez se terminó todo este proceso, Grilla llama al método ejecutarEfectos() de la Clase Bloque, particularmente de la instancia donde "quedó" Pacman

¿Qué es un Esperador?:

Un Esperador es una suerte de implementación del patrón Observer, ya que es un objeto cuyo propósito es notificarle algo a otro. Decimos que es "una suerte de implementación" porque se le suma que el Esperador espera intervalo milisegundos antes de notificarle a Grilla algo.

Consideremos el caso del EsperadorBomba. Su funcionamiento es, más o menos, como sigue:

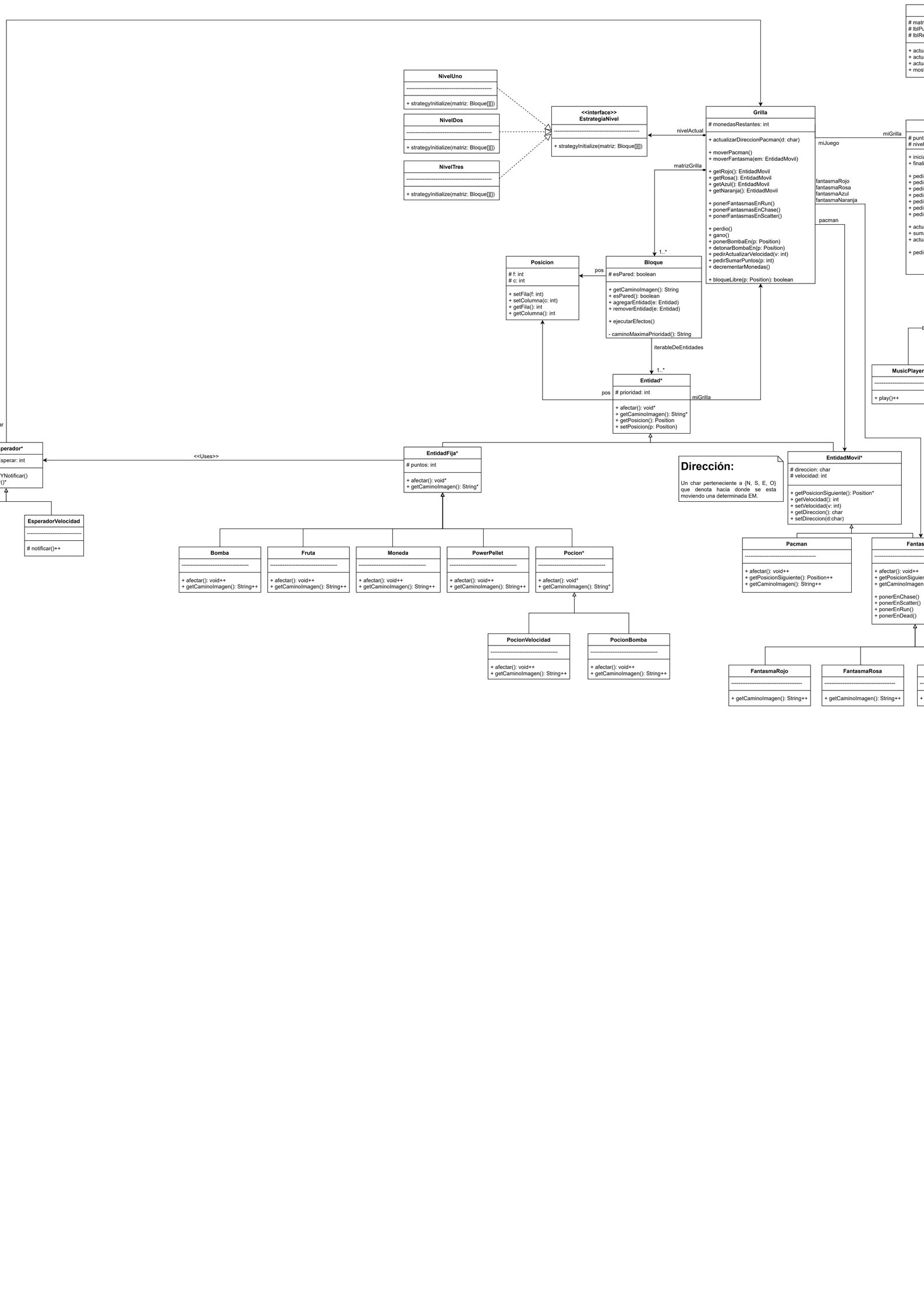
1. Pacman "toma" una pución bomba.
2. Se agrega una entidad Bomba al bloque en cuestión.
3. Se crea un EsperadorBomba con 5000ms de intervalo.
4. Pasan esos 5000ms
5. El EsperadorBomba le notifica a Grilla que es hora de detonar la bomba. Ese hilo muere inmediatamente después.
6. Grilla efectúa la detonación.

Patrón Template:

Esperador redefine el método run() de Thread con la siguiente estructura:

```
run() {
    Thread.sleep(tiempoAEsperar);
    notificar();
}
```

Donde notificar es un método abstracto, y es el que redefinen las clases hijas. El método run() como tal es la Template.



Recursos:

ResourceHandler aplica el patron Strategy para mostrar las diferentes "skins" del juego, delegando en las implementaciones de su interfaz SkinInterfaz.

El resto de las clases interactuan con ResourceHandler, y este sera estático (y quizás Singleton)

CI Significa "Camino Imagen"

Launcher

+ main(String[] args)
+ lanzarMenu()
+ lanzarJuego(nivel: int)

miJuegoGUI
miMenuGUI
<<Uses>>
miJuego

ResourceHandler

+ getParedCI(): String
+ getPisoCI(): String

+ getPacmanCI() : String

+ getFantasmaRojoCI(): String
+ getFantasmaRosaCI(): String
+ getFantasmaAzulCI(): String
+ getFantasmaNaranjaCI(): String

+ getMonedaCI(): String
+ getFrutaCI(): String
+ getPowerPelletCI(): String

+ getPocionVelocidadCI(): String
+ getPocionBombaCI(): String

skinElegida
1

<<Interface>>
SkinInterface

+ getParedCI(): String
+ getPisoCI(): String

+ getPacmanCI() : String

+ getFantasmaRojoCI(): String
+ getFantasmaRosaCI(): String
+ getFantasmaAzulCI(): String
+ getFantasmaNaranjaCI(): String

+ getMonedaCI(): String
+ getFrutaCI(): String
+ getPowerPelletCI(): String

+ getPocionVelocidadCI(): String
+ getPocionBombaCI(): String

SkinStarWars

SkinCarpincho

SkinTetris

GUIs:

Launcher tiene el metodo main. En main, se crea una MenuGUI, donde se puede seleccionar nivel y skin. Cuando se selecciona, se aprieta "Jugar" y eso le manda un mensaje a Launcher, que cierra MenuGUI y abre JuegoGUI

miLauncher
miLauncher

JuegoGUI

izLabels JLabel[][]
ntaje: JLabel
lo): JLabel

---

alizar(p: Position, ci: String)
alizarPuntaje(p: String)
alizarTiempo(t: String)
trarPerdio()

MenuGUI

skinElegida: String
nivelElegido: int

+ pedirLanzarJuego()

miJuego

Thread

/----------/

/----------/

Extends

miGUI

Juego

aje: int
: int

---

rPartida()
zarPartida()

rActualizar(p: Position, ci: String)
rMoverPacman()
rMoverRojo()
rMoverRosa()
rMoverAzul()
rMoverNaranja()
rEstadoMusica(estado: boolean)

alizarPuntaje(p: int)
arPuntos(p: int)
alizarVelocidad(v: int)

rActualizarDireccionPacman(d: char)

1

miJuego

Reloj*

intervalo: int
ciclo
sePuedeJugar:boolean

+ setIntervalo(intervalo: int)
+ setSePuedeJugar(b: boolean)

relojFantasma
relojPacman

Extends
Extends

RelojFantasmas

+ run()++

RelojPacman

+ run()++

miSoundPlayer
miMusicPlayer
1
1

AudioPlayer*

caminoSonido : String

+ play()*

SoundPlayer

+ play()++

ma*

---

te(): Position++
(): String*

1 miEstado

<<interface>>
EstadoFantasma

+ siguientePosicion(): Position
+ afectar()

FantasmaCeleste

---

getCaminoImagen(): String++

FantasmaNaranja

+ getCaminoImagen(): String++

Chase

+ Chase(ia: ChaseIA)
+ siguientePosicion(...)++
+ afectar()++

Scatter

posicionBuscada: Position

+ Scatter(eb: Position)
+ siguientePosicion(...)++
+ afectar()++

Run

+ siguientePosicion(...)++
+ afectar()++

Dead

+ siguientePosicion(...)++
+ afectar()++

IAMovimiento

ChaseIA*

+ calcularSiguientePosicion(): Position*

Chase IA:

El Estado Chase delega en su clase Chase IA, la cual es especificada por las diferentes IAs de color.

La idea de esto es mantener la genericidad lo más posible y no tener un Estado chase por cada color.

Es una suerte de Bridge.

ChaseRojo

+ calcularSiguientePosicion(): Position++

ChaseRosa

+ calcularSiguientePosicion(): Position++

ChaseAzul

+ calcularSiguientePosicion(): Position++

ChaseNaranja

+ calcularSiguientePosicion(): Position++