Práctica 8: Come-cocos.

Cristian Alfonso Prieto Sánchez.

D.N.I.: 75926381-T.

Correo electrónico: cristian92@correo.ugr.es .

Rafael González Callejas

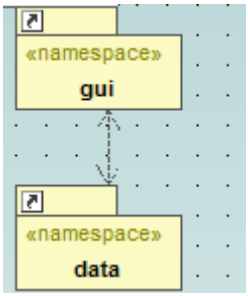
D.N.I.: 26149480-W.

Correo electrónico: callejas1992@correo.ugr.es .

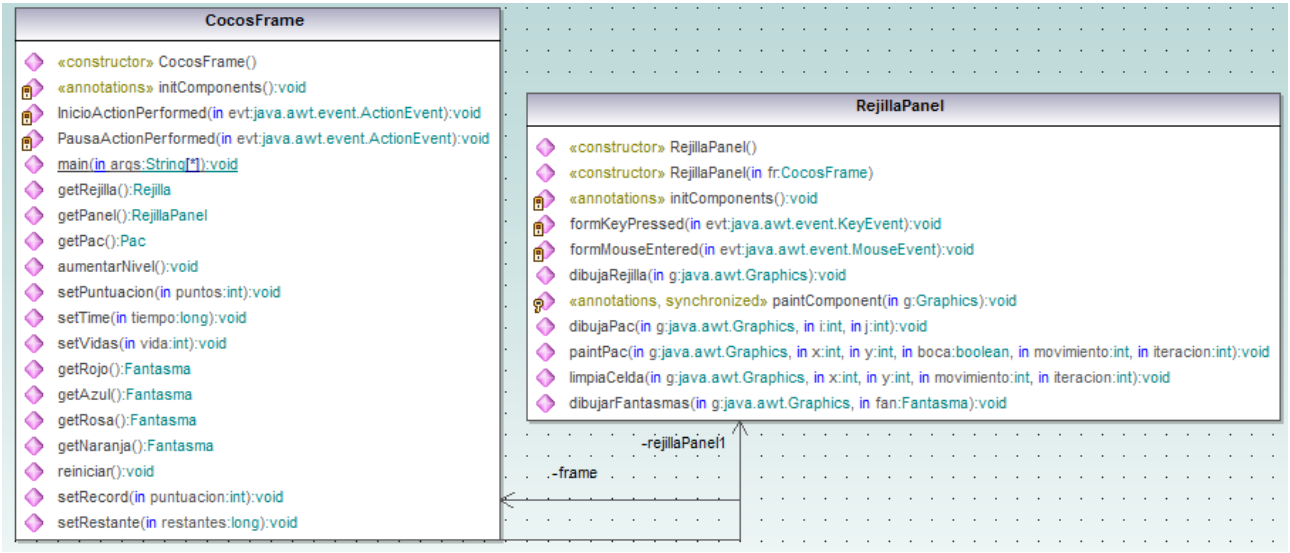
Índice de contenidos:

1. Diagrama UML:

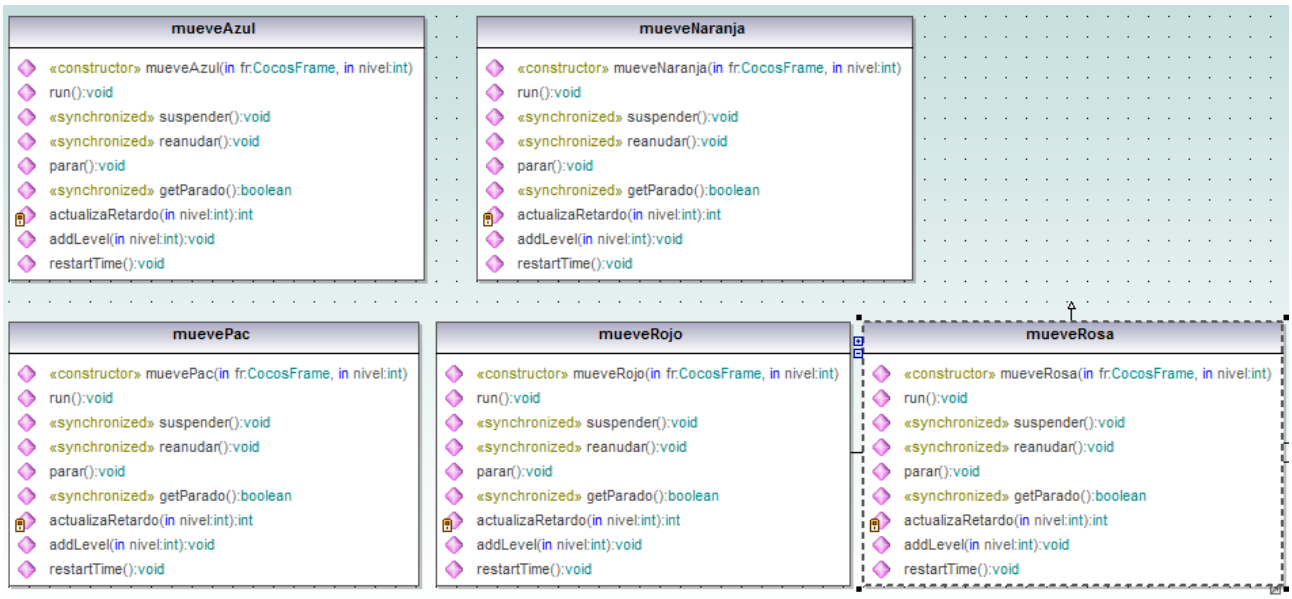
Relación entre los paquetes.

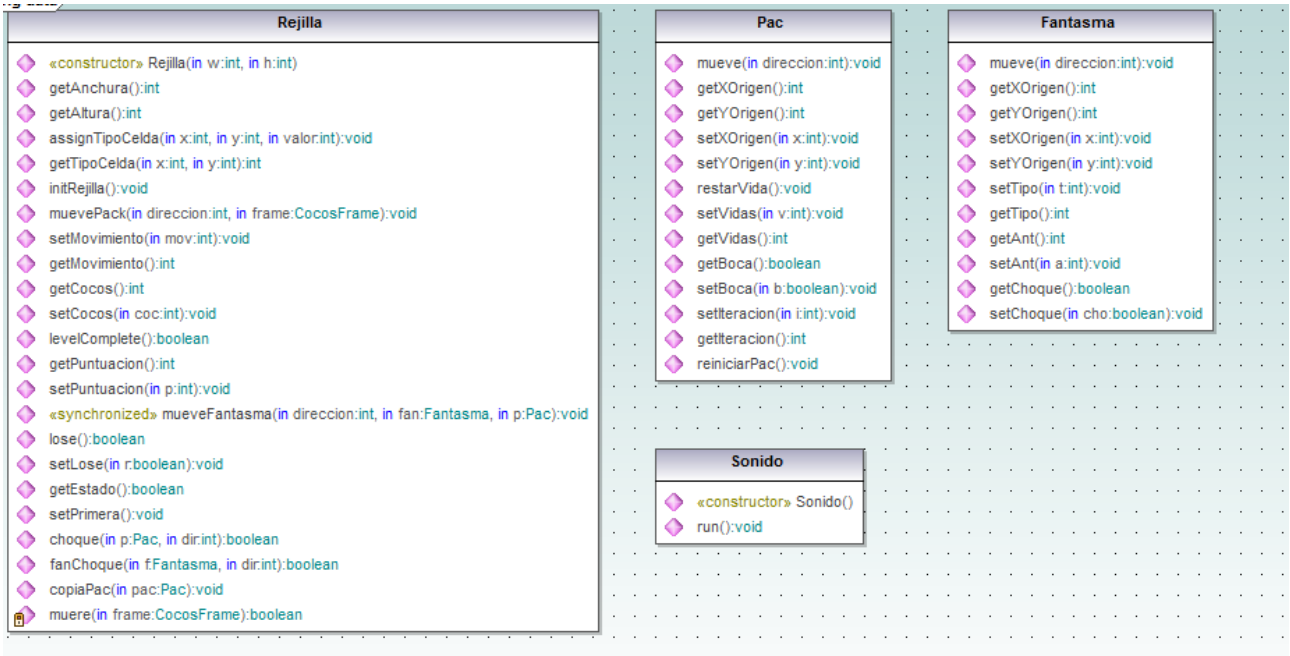


Relaciones del paquete gui.



Relaciones en el paquete data.





2. Solución:

Comenzaremos a analizar nuestro programa a partir de las clases más sencillas y menos abstractas, en concreto, podríamos comenzar con las clases Rejilla, Pac y Fantasma, que son clases que contienen datos y métodos sobre los que se va a construir todo el comecocos.

A continuación subiremos escalones en el nivel de abstracción hasta llegar finalmente a la clase CocosFrame, la clase principal del proyecto.

2.1. Rejilla.

La clase Rejilla es una clase bastante sencilla, contiene datos relativos a los elementos que forman parte del laberinto (bloques, fantasmas, comecocos, dimensiones del laberinto, numero de cocos, estado de los fantasmas, etc), así como métodos que permiten modificar el valor de dichos datos.

Los métodos de esta clase son los siguientes:

- Un constructor Rejilla() que crea una rejilla de dimensiones (w, h), reservando su espacio en memoria.
- Los métodos getAnchura() y getAltura() que se utilizan para obtener la anchura o la altura de la rejilla, respectivamente.
- Los métodos assignTipoCelda() y getTipoCelda() se utilizan para asignar a una celda determinada el valor que queramos, o para obtener el valor que en un momento dado tiene la celda.
- El método initRejilla() se utiliza para inicializar la rejilla con valores definidos por el usuario, en cada posición se coloca o bien un tipo de bloque que forma parte del laberinto, una celda que forma una calle con coco, o con un coco especial, un fantasma, o el propio come-cocos.
- El método muevePack() trabaja a partir de una rejilla ya construida, en los primeros movimientos se encarga de cambiar las celdas que contienen fantasmas por celdas vacías. Además, en función del parámetro di-

rección estudiamos la siguiente casilla a la que se va a mover el come-cocos y colocamos las condiciones necesarias para que éste no se desplace a una casilla que contiene un bloque o, en caso de que se desplace a una casilla que tenga un fantasma, si estos no se pueden comer, se pierda una vida, además, al pasar por una celda con coco, se aumenta la puntuación. Algunos parámetros importantes son el número de cocos comidos, usado para aumentar el nivel de la partida, o el contador de movimientos que puede realizar un fantasma antes de dejar de ser comestible.

- El método mueveFantasma() trabaja de forma similar a muevePack(), su objetivo en este caso es controlar el movimiento de los fantasmas, para lo cual se necesita saber el tipo de fantasma con el que estamos trabajando, su posición de origen y la dirección hacia la que se dirige. Controlaremos entre otras cosas, el tipo de celda en la que se encontraba el fantasma y el tipo de celda a la que se dirige, así como la posibilidad de que el fantasma sea comido por el come-cocos, sumando 200 puntos al marcador.
- Los métodos setMovimiento() y getMovimiento() se utilizan para almacenar el movimiento que ha realizado el come-cocos y obtener dicho movimiento respectivamente.
- getCocos() y setCocos() obtienen y modifican el número de cocos que se han comido.
- El método levelComplete() indica si se han dado las condiciones necesarias para pasar de nivel.
- getPuntuacion() y setPuntuacion() devuelven y modifican la puntuación de la partida.
- lose() y setLose() controlan la condicion de derrota.
- getEstado() informa si el come-cocos persigue a los fantasmas o si por el contrario son los fantasmas los que persiguen al come-cocos.
- El método setPrimera() indica si nos encontramos en los primeros movimientos de la partida, en los que deben cumplirse unas condiciones especiales que dependen del valor devuelto por este método.
- Choque() y fanChoque() son otros de los métodos importantes de la clase Rejilla, se encargan de controlar si va a producirse un choque a partir de la dirección del come-cocos o del fantasma que se este controlando y de las coordenadas actuales de éste, devuelven true si no va a producirse un choque y false en caso contrario.
- copiaPac() se encarga de cambiar el tipo de casilla a la que se vaya a desplazar el come-cocos por el tipo P.
- Muere() controla si el come-cocos chocará con un fantasma y perderá una vida.

2.2. Clase Pac.

La clase Pac es una clase más sencilla que la clase Rejilla, básicamente esta clase controla los parámetros necesarios para controlar el come-cocos. Como datos miembro contamos con cuatro variables que controlan las direcciones en las que se puede mover el come-cocos, así como sus coordenadas de origen iniciales, el número de vidas, el estado de la boca y la iteración en la que se encuentra, para cuando controlemos el movimiento del come-cocos con cuatro iteraciones.

La mayoría de los métodos de esta clase se encargan de obtener o modificar los datos miembros de la clase, estos métodos son: getXOrigen() y getYOrigen(), setXOrigen() y setYOrigen(), restarVida(), setVidas() y getVidas(), getBoca() y setBoca() o setIteracion() y getIteracion().

Existe un método diferente a los anteriores, más básicos, encargado de modificar la posición del come-cocos, este método mueve() modifica las coordenadas xorigen o yorigen en función del parámetro de entrada, direccion.

2.3. Clase muevePac.

En la clase muevePac empezamos a utilizar métodos mas completos, incluyendo hebras, retardos, y elementos que hemos reutilizado de proyectos anteriores, sobre todo del tetris realizado en la práctica 7.

La clase muevePac, que implementa la interfaz Runnable, cuenta con una serie de datos miembros necesarios para el correcto funcionamiento de nuestro come-cocos, entre las que se encuentran un retardo "delay", las variables "continuar" y "suspendflag" que controlan si la hebra debe o no seguir funcionando o esperar, una referencia a nuestro frame, un indicador del nivel en el que nos encontramos y la variable "time_start" que controla el momento de comienzo del juego. También hay un tipo de dato long que muestra una referencia al tiempo de comienzo de cada nivel específico y otro dato con referenci al tiempo máximo por nivel.

El constructor muevePac() se encarga de iniciar la referencia al frame, así como de inicializar la variable que controla el nivel, el retardo, y la hebra que se encarga de mover el come-cocos.

El método run() es el método donde incluimos las sentencias necesarias para que el come-cocos se mueva. Dentro se gestionan elementos como el número de vidas, el sonido que se escuchará cuando el come-cocos es comido por un fantasma, o el siguiente movimiento que va a realizar el come-cocos, además se usan métodos que ya hemos comentado anteriormente y que definimos en la clase rejilla y en la clase pac. Para garantizar el funcionamiento del juego, se han de gestionar dos excepciones, una referente al sonido del juego y otra referente a la interrupción del movimiento del Pacman en caso de que este sea comido por un fantasma.

Existen otra serie de métodos que se encargan de gestionar el funcionamiento de la hebra, estos métodos son:

- suspender(): Detiene la ejecución de la hebra haciendo momentáneamente.
- reanudar(): Reanuda el movimiento de la hebra.
- parar(): Termina la ejecución de la hebra.
- getParado(): Informa si la hebra está (true) o no parada (false).

El método actualizaRetardo() controla la velocidad con que se van a mover algunas figuras en función del nivel en el que nos encontremos.

El control del nivel en el que nos encontramos se realiza a través del método addLevel(), donde también se añade a la puntuación total unos puntos extra en función del nivel y de si se ha conseguido acabarlo antes del tiempo referencia.

Finalmente el método restartTime() actualiza el tiempo de inicio de partida al valor actual.

2.4. Clase RejillaPanel.

La clase RejillaPanel es, junto con la clase cocosFrame, la clase más abstracta del proyecto, para su generación hemos seguido los pasos descritos en el guión de la práctica siete, generando varios métodos automáticamente siguiendo dichos pasos, sirviéndonos de las herramientas que NetBeans pone a nuestro alcance para facilitar dicho proceso.

Si empezamos a observar esta clase, observamos que contiene tres datos miembro, una referencia a nuestro frame, la variable anchoCelda, que nos servirá para dibujar correctamente los elementos que formarán parte del come-cocos y la variable pausa, que controlará el funcionamiento del juego.

Los constructores RejillaPanel() se encargar de generar un nuevo formulario RejillaPanel, son métodos que se generan automáticamente al crear un nuevo form JPanel y que modificamos adecuadamente de acuerdo a como se indica en la práctica 7.

Por su parte el método initComponents() es un método que se encarga de generar todos los elementos que forman parte de nuestra rejilla y que hemos seleccionado en los menús de elementos del paquete swing de java.

Para controlar el movimiento del come-cocos a través de las teclas del teclado se debe generar el método formKeyPressed(), en él se asigna cada tecla de las cuatro flechas de dirección a un movimiento (de 0 a 3) que previamente hemos asignado a una dirección concreta (consultar clase Pac o clase Fantasma).

Contamos con el método fromMouseEntered(), que funcionaba con el objetivo de no iniciar el juego hasta que se pasase el ratón por el panel de juego, ahora no es necesario ya que al pulsar el botón iniciar el juego inicia y el foco pasa al panel directamente.

Por su parte, el método dibujaRejilla(), accede a la Rejilla del frame para consultar el tipo de celda que se encuentra en cada posición de la matriz que forma el laberinto y todos los elementos de nuestro come-cocos, dibujando en cada caso el elemento indicado.

El método paintComponent() dibuja todo elemento gráfico, tanto la rejilla como el come-cocos y los fantasmas que posteriormente generaremos.

El método dibujaPac() hace uso del paquete graphics de Java y de dos variables, i y j que permiten dibujar el come-cocos en la celda que le corresponda en función del movimiento que se esté realizando en cada instante.

El resto de métodos de esta clase forman parte del apartado de mejoras y se comentarán en el apartado siguiente con más detalle.

2.5. Clase CocosFrame.

Finalmente, la clase CocosFrame constituye la clase principal de nuestro proyecto, en ella, al igual que sucedía en la clase RejillaPanel, se generan varios métodos de forma automática al seleccionar y colocar elemento del menú swing de java en nuestro JFrame.

En primer lugar analizando la clase CocosFrame observamos todos los datos miembros que posee, una referencia a cada clase que forma parte de nuestro proyecto y algunas variables que controlan el desarrollo del mismo, como son: nivel, pausa y primera.

A continuación se encuentra el constructor de la clase, CocosFrame(), que se encarga de crear el formulario JFrame, inicializando todos los componentes que forman parte de él y las clases que gestionarán el funcionamiento de los elementos que constituyen el juego.

InitComponents() se encarga de generar todos y cada uno de los elementos que van a formar parte de nuestro Frame, da forma y ordenada cada elemento de manera que al iniciar el juego cada elemento se encuentre situado donde nosotros previamente lo hemos colocado.

Los métodos InicioActionPerformed() y PausaActionPerformed() usan las condiciones primera y pausa respectivamente para o bien iniciar los primeros pasos del juego o bien pausar y reanudar este.

Dentro del main() se gestionan una serie de excepciones relativas al funcionamiento de los elementos que forman parte del frame y que se han generado al crear nuestro frame haciendo uso de los elementos obtenidos del paquete swing de java.

Finalmente quedan por comentar una serie de métodos bastante sencillos, entre los que se encuentran getRejillaPanel(), getPanel(), getPac(), que se encargan de obtener las referencias a sus respectivas clases.

También hay algunos métodos que controlan aspectos secundarios del juego, como el incremento del nivel del mismo, a través del método aumentarNivel(), la puntuación, controlada por el método setPuntuación(), el tiempo de juego en cada nivel, a través de setTime(), o las vidas restantes, controladas por el método setVidas();

Existen de igual forma referencias a los distintos fantasmas que van a formar parte del come-cocos en la parte de mejoras y un método que se encarga de gestionar el reinicio de la partida, en el que todos los elementos del juego deben volver a su estado inicial, reiniciar().

3. Mejoras implementadas.

Vamos a tratar ahora de explicar el conjunto de mejoras que hemos tratado de implementar en nuestro juego del come-cocos, si bien, al tratarse de un programa tan extenso, resulta un tanto lioso comentar algunas mejoras, que se encuentran en partes muy puntuales del programa y que puede que inevitablemente ya hayamos comentado a lo largo del apartado anterior. Con ello nos referimos por ejemplo a las siguientes mejoras:

- Mejora 1: Al generar nuestro laberinto para el comecocos, ya hemos incluido las celdas que corresponden a los puntos pequeños y grandes, y dentro de la propia clase Rejilla ya hemos incluido variables y métodos que se encargan de gestionar esta mejora, como pueden ser las variables cocos, numCocos o puntuación y los métodos getCocos(), setCocos(), getPuntuacion(), setPuntuacion(), etc.

  Además las variables cocos y numCocos ya se han usado en otros métodos por ejemplo el método levelComplete() que sirve para indicar que hemos completado un nivel, ya que hemos comido todos los cocos del mismo. Como salvedad cada vez que morimos se reinician los cocos que hay en la rejilla, pero esta variable no, pudiendo subir de nivel aunque no se hayan comido todos los que se ven por pantalla.

- Mejora 2: Esta mejora también la hemos implementado y comentado en el apartado anterior, basta con observar la matriz que forma nuestra rejilla para comprobar que existen diferentes tipos de bloques que permiten redondear los diferentes conjuntos de bloques y hacer que el laberinto del come-cocos tenga un aspecto más amigable, si bien ello exige el incremento en el número de condiciones de otros métodos como por ejemplo el método dibujaRejilla(), de la clase RejillaPanel.

- Mejora 3: La forma de detener y reanudar el juego, a través de botones en el panel de juego ya se ha comentado también a lo largo del desarrollo del guión, así como los métodos que se encargan de generar esos eventos, por ejemplo en el ultimo apartado de la sección dos se han comentado los métodos reanudar() o PausaActionPerformed(), que gestionan esta mejora.

  Comentar que el botón pausa se encarga de controlar este evento y hace que si se pulsa pausa se pause el juego, y la segunda vez que se pulsa, se renauda el juego.

- Mejora 7: También hemos usado la condición de pasar de nivel cuando el come-cocos se coma todos los cocos de un nivel, ya en la mejora 2 hemos comentado la utilización del método levelComplete() que utiliza el número de cocos comidos y el número de cocos totales para establecer cuando se ha completado un nivel y se puede pasar al siguiente, incrementando la velocidad de movimiento de los distintos elementos del juego. Además se reinicia la variable de tiempo restante ya comentada.

- Mejoras 10 y 11: Se ha añadido también la opción de guardar un récord del juego. Para ello ha sido necesario añadir el método setRecord() en la clase cocosFrame. Este método que recibe como parámetro un entero, leerá el fichero “record.txt”, si existe leerá el primer dato y comparará la puntuación dada con el dato leído, si la puntuación es mayor a la leída, tendremos nuevo récord y se sobreescribirá el antiguo. Por último en un área de texto se imprimirá el nuevo record.
- Mejoras 13 y 14: Se ha añadido en la clase Pac un dato miembro referencia y otro referencianivel que contienen el tiempo máximo para los niveles permitido y el tiempo de inicio de cada nivel, respectivamente. Con esto conseguimos saber el tiempo transcurrido en cada nivel permitiéndonos tener una referencia clara. En vez de acabar con una vida cuando se acaba el tiempo lo que hemos programado es que al acabarse el tiempo no se obtenga bonificación de puntos al pasar de nivel, mientras que si acabamos un nivel antes del tiempo de referencia, obtendremos una bonificación proporcional al nivel en el que nos encontremos.

Hay otra serie de mejoras que conviene analizar con mayor detenimiento, pues requieren la creación de clases dedicadas o necesitan la modificación de una mayor cantidad de código.

3.1. Mejora 4: Movimiento más fluido del Come-cocos.

La mejora del movimiento del come-cocos se realiza en el método paintPac() de la clase RejillaPanel, que hemos explicado con detenimiento en el apartado anterior. Para conseguir una mejora en la movilidad realizamos el movimiento de una celda a la siguiente en un total de cuatro iteraciones, a través de la inclusión de un nuevo contador, que indica la iteración en la que nos encontramos, básicamente lo que se consigue es pintar el come-cocos (a través del método paintPac()) en cuatro posiciones en las que dividiríamos cada celda en cuatro subceldas, dividiendo entre cuatro la anchura total de la celda y multiplicandola por la iteración en la que se encuentre en cada caso.

Este proceso que requiere cuatro iteraciones para cada movimiento del come-cocos debe realizarse con un retardo cuatro veces más pequeño del que tiene la hebra que ejecuta nuestro frame.

Además en cada iteración se debe controlar la orientación del come-cocos y la posición de su boca, de forma que se puede implementar la mejora que vamos a comentar a continuación.

La mejora no se incluye inicialmente ya que no está bien depurada y el movimiento no es tan perfecto como se desearía, además el dibujo realizado en paintPac() no es del todo correcto y provoca una sensación de falta de trabajo que no se desea transmitir. Sin embargo si queremos probar esta mejora solo debemos descomentar en el método paintComponents() de RejillaPanel la llamada a paintPac() (y eliminar dibujaPac()) y descomentar en muevePac lo necesario para realizar el movimiento en 4.

3.2. Mejora 5: Diferentes aspectos para el come-cocos en función de su posición.

Para mejorar el aspecto del come-cocos, consiguiendo que su orientación cambie de acuerdo a como lo hace su movimiento y su boca se abra y se cierre de forma alternativa vamos a utilizar los métodos getBoca() y getIteracion() de la clase Pac, que nos permitirán pintar para cada movimiento del come-cocos la posición de su boca (abierta o cerrada) y nos indicará la orientación del mismo a través del método getMovimiento(), dentro del método comentado en el apartado anterior, paintPac(), y también en el método usado actualmente dibujaPac(), ambos de la clase RejillaPanel.

3.3. Mejora 6: Generación y control de los cuatro fantasmas.

Para gestionar el funcionamiento de los cuatro fantasmas dentro del comecocos necesitamos llevar a cabo un proceso parecido al llevado a cabo para gestionar el funcionamiento del come-cocos, con la excepción de que en este caso no vamos a controlar la posición del elemento en funcion de su desplazamiento, ni tampoco tendremos boca para controlar.

En principio necesitaremos una clase, Fantasma, que contendra todos los datos y metodos relativos a los niveles más básicos de funcionamiento de un fantasma, entre los que se encuentran, la dirección hacia la que se va a mover, el color del fantasma, si son vulnerables para ser comidos por el come-cocos, así como las coordenadas de origen de cada fantasma y una variable ant en la que guardamos el valor de la celda anterior en la que se encontraba el fantasma que estemos gestionando.

Dentro de los métodos básicos para gestionar a cada fantasma se encuentran los métodos mueve(), getXOrigen, getYOrigen(), setXOrigen(), setYOrigen(), setTipo(), getTipo(), getAnt(), setAnt() y getChoque() y setChoque().

No hacemos especial hincapié en estos métodos porque son identicos a los usados en la clase Pac, donde obteníamos prácticamente los mismos parámetros que posteriormente nos permitirán obtener la información necesaria de cada fantasma para poder controlar todas las posibles situaciones que tengan lugar durante el desarrollo del juego.

Para controlar el movimiento de cada fantasma vamos a usar una hebra por cada uno, por ejemplo, el movimiento del fantasma de color azul se gestiona en la clase mueveAzul. Dentro de esta clase nos encontramos con los datos miembros básicos, el retardo, la bandera que gestionará el funcionamiento de la hebra, la referencia al frame del comecocos, una variable que gestiona el nivel y una variable que nos sirve para controlar el tiempo de inicio de la hebra.

Dentro del método run() cada fantasma se encarga de moverse de acuerdo a un razonamiento, en caso de no estar el tiempo especial (tiempo durante el cual el pac puede comer un fantasma), que es el siguiente. Dependiendo de la posición del pac el fantasma intentará ir hacia él por el camino más cercano, si el pac está arriba a la izquierda, el fantasma podrá ir a izquierda o arriba, con la misma probabilidad para cada

camino. El fantasma tomará uno de estos dos caminos, si solo uno es posible tomará el posible. Si durante cinco intentos el fantasma no ha sido capaz de moverse en ninguna dirección, entonces el fantasma se moverá en una de las dos direcciones restantes y descartadas al principio. Esto último se hace solo para evitar que el fantasma se quede atrancado en una posición.

El resto de métodos ya los hemos visto con anterioridad, básicamente se encargan de parar y reanudad la hebra que estamos gestionando modificando el valor de suspendFlag (métodos reanudad() y suspender()) o terminar la ejecucion de la hebra (método parar()). También se encargan de informar sobe el estado de la misma (getParado()), actualizar el retardo (actualizaRetardo()), aumentar el nivel de juego de la partida (addLevel()) y modificar el valor del tiempo que se lleva jugado.

3.4 Mejora extra: Sonido de juego.

Como última mejora se ha creado una nueva clase que implementa Runnable, llamada sonido, y que contiene una hebrá que hará sonar música durante el juego con el fin de amenizar el juego.