uc3m | Universidad Carlos III de Madrid

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA INFORMÁTICA

PROYECTO FINAL - PROGRAMACIÓN

MINI GALAGA

ALUMNOS: IVÁN MARTÍN Y MANUEL MUÑOZ GÓMEZ

TUTOR: JOSÉ CARLOS GONZÁLEZ DORADO

ÍNDICE

INTRODUCCIÓN

El proyecto que se presenta consiste en la realización del videojuego "*Galaga*" mediante el lenguaje de programación *Java* y una interfaz gráfica proporcionada por el departamento de la asignatura.

Dentro del proyecto, se encuentran las clases necesarias para el correcto funcionamiento de este. La clase "***Sounds***" ha sido reciclada desde la web, y gracias a ella hemos podido implementar sonidos. La clase "***GameBoardGUI***" nos ha sido facilitada para poder desarrollar el juego correctamente.

Por otra parte, hemos llevado a cabo el desarrollo de imágenes como el fondo del juego, el SuperCommander (Boss final), etc., que mejoran el aspecto visual de este.

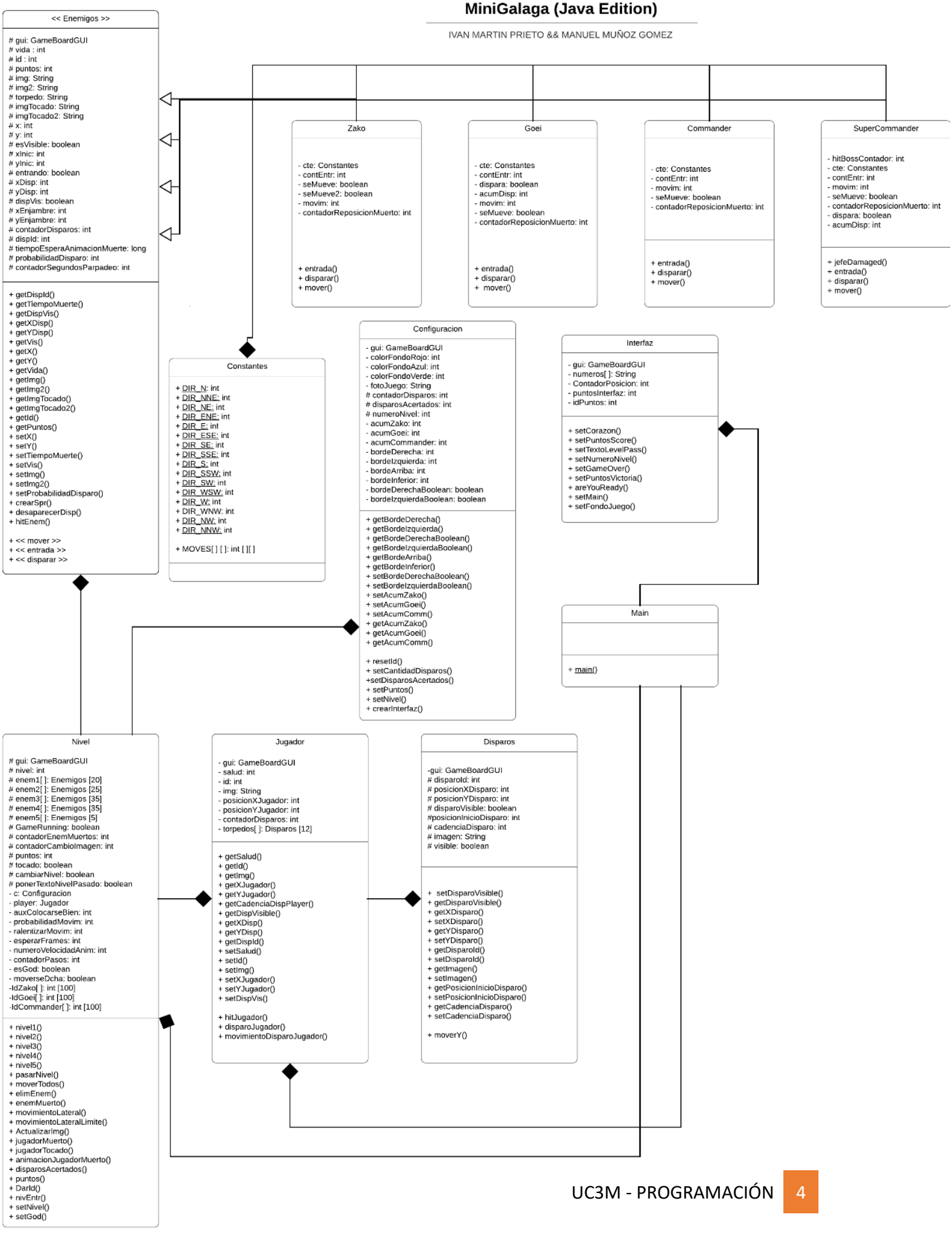
DISEÑO DE CLASES

A continuación, se presenta un diagrama de clases en el que se detalla la estructura del proyecto. En total hemos diseñado 12 clases (sin contar la clase "*Sounds*" y "*GameBoardGUI*").

(*ver página siguiente)*

MiniGalaga (Java Edition)

IVAN MARTIN PRIETO && MANUEL MUÑOZ GOMEZ



ALGORITMOS PRINCIPALES

A lo largo del proyecto, hemos tenido que realizar distintos métodos con el objetivo de cumplir las condiciones propuestas. A continuación, enumeramos aquellos que nos han parecido más relevantes:

- DarId (String tipo): Este método nos proporciona mayor facilidad a la hora de asignar id's a los distintos Sprites dependiendo del tipo que sean. Este método es apoyado por otros métodos como getAcumZako(), getAcumGoei() o getAcumComm().
  Su funcionamiento consiste en devolver una posición de un array creado anteriormente cuyos valores de las posiciones son números enteros que van del 100 al 199 (zako), 201 al 300 (goei), y 301 al 400 (commander). A través de un *switch*() selecciona el tipo de enemigo del que devuelve el valor del array, en el caso por defecto (no debería ocurrir nunca) devuelve un número predefinido. Mediante un acumulador conseguimos que la posición que devuelve se incremente. (**Muestra del código**):

```
public int DarId(String tipo) {

	switch (tipo) {
	case "zako":
		c.setAcumZako();
		return IdZako[c.getAcumZako()];

	case "goei":
		c.setAcumGoei();
		return IdGoei[c.getAcumGoei()];

	case "commander":
		c.setAcumComm();
		return IdCommander[c.getAcumComm()];

	default:
		return 12736; // Numero cualquiera [nunca lo devuelve]

	}
}
```

- moverTodos (Enemigos[ ] arrEnem): Mediante este algoritmo conseguimos agrupar los distintos movimientos que realizan los enemigos en distintos niveles, aunque también se ocupa de evitar que estos movimientos se realicen demasiado rápido.
  Su funcionamiento se basa en generar un número aleatorio con el que se llama a otro método (mover *())*, este retoma el valor del número generado, y si coincide con el valor asignado a cada sprite, este realiza un movimiento descendente tras el cual regresa a su posición en el enjambre. La potencia de este método se ve cuando se genera un número aleatorio común a todos los Sprites permitiendo el movimiento simultáneo de estos.

```
public void moverTodos(Enemigos[] arrEnem) {
		if (System.currentTimeMillis() - ralentizarMovim >= 25) {
			probabilidadMovim = ((int) (Math.random() * 1000 + 1));
			ralentizarMovim = System.currentTimeMillis();// para que
vayan despacio
			if (nivel < 4) {
				for (int i = 0; i < arrEnem.length; i++) {
```

```
                        arrEnem[i].mover(probabilidadMovim, false);
                    }
                }
                if (nivel == 4) {
                    for (int i = 0; i < arrEnem.length; i++) {
                        arrEnem[i].mover(probabilidadMovim, true);
                    }
                }
                if (nivel == 5) {
                    for (int i = 0; i< arrEnem.length; i++) {
                        arrEnem[i].mover(probabilidadMovim, false);
                    }
                }
            }
    }
```

- nivEntr(): Con este algoritmo hemos querido evitar que se realice cualquier acción mientras los enemigos están entrando.
  Su funcionamiento consiste en comprobar uno a uno si todos los enemigos del array del nivel en el que te encuentras están realizando la animación de entrada. En caso de que uno se encuentre realizando dicha animación devolvería el valor "true", en caso de que ninguno la esté realizando devolvería el valor "false". (**Muestra del código**):

```
public boolean nivEntr() {

    if (nivel == 1) {
        for (int i = 0; i < enem1.length; i++) {
            if (enem1[i].entrando == true)
                return true;
        }
    }

    if (nivel == 2) {
        for (int i = 0; i < enem2.length; i++) {
            if (enem2[i].entrando == true)
                return true;
        }
    }
    if (nivel == 3) {
        for (int i = 0; i < enem3.length; i++) {

            if (enem3[i].entrando == true)
                return true;
        }
    }
    if (nivel == 4) {
        for (int i = 0; i < enem4.length; i++) {

            if (enem4[i].entrando == true)
                return true;
        }
    }
    if (nivel == 5) {
        for (int i = 0; i < enem5.length; i++) {

            if (enem5[i].entrando == true)
                return true;
        }
    }

    return false;}
```

- entrada(int k, Enemigos[ ] arrEnem): Este método funciona mediante una variable *boolean* ("entrando") cuyo valor se inicializa "true" al comienzo de cada nivel. Si es "true" comienza a sumar posiciones en ambos ejes de coordenadas de los enemigos. Sus direcciones son obtenidas de la clase *Constantes* y sus imágenes van cambiando, dependiendo de su orientación.
  Su funcionamiento se basa en la aplicación de un sistema de ejes coordenados con sus respectivas direcciones. De esta manera, los enemigos realizan una coreografía sobre el tablero hasta recolocarse en su posición de inicio. (*El fragmento de código es demasiado extenso para presentarlo aquí, pero puedes verlo en la clase de cada enemigo [Goei, Commander, Zako Y SuperCommander])*

FUNCIONALIDAD IMPLEMENTADA

- SPRINT 1: (Enjambre, Jugador básico)

Enemigos:

La estructura del enjambre ha sido creada mediante un array del tipo *Enemigos*. Dentro del constructor de esa clase hemos asignado a cada Sprite una posición inicial colocándose de manera que simulan una formación.

En el nivel 1 aparecen los enemigos *Goei* y *Zako*. A partir del nivel 2, hemos implementado el enemigo *Commander*, el cual ha sido modificado en el nivel 5 para poder definir un *Boss final*. La imagen de este enemigo ha sido redimensionada, y su vida se ve aumentada.

Enemigo nivel 5

Jugador:

El Sprite del jugador es creado en el *Main* y su movimiento es recogido por un *switch( )* que mediante el método "gui.gb_getLastAction()" selecciona qué dirección se ha pulsado. Dependiendo de esta dirección, su posición en el eje X se ve modificada. Si esta dirección supera el 165 o es inferior al 5, no se realiza ningún movimiento, manteniendo al jugador en el borde del tablero.

Su velocidad de movimiento es implementada por el método "gui.gb_moveSpritecoord()" que permite el movimiento del jugador en décimas de casilla.

Dentro del constructor del jugador, le hemos asignado ciertas características como "salud". El resto de los atributos presentes en el marcador los hemos implementado dentro de la clase "Nivel".

- SPRINT 2: (Animaciones, Torpedos del jugador)

Enemigos:

El movimiento del enjambre ha sido realizado mediante métodos dentro de la clase "nivel". En estos métodos se modifica el valor de la X de cada enemigo del enjambre hasta que llegan a uno de los bordes, donde mediante un "boolean" reciben la orden de aumentar su valor de la Y, cambiando el sentido de movimiento en el eje X.

Cada x frames, la imagen de cada enemigo se ve modificada, simulando un movimiento de "aleteo". Cada ciclo que realiza el bucle, el método "actualizarImg()" encontrado en la clase "Nivel" cambia la imagen de cada Sprite por la que tiene asignada.

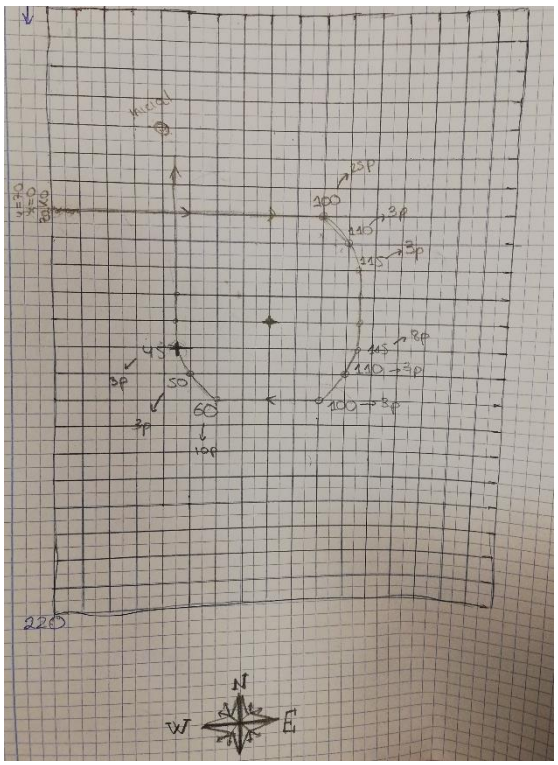
Jugador:

Dentro del "Main" hemos implementado un *switch()* que mediante el método "gui.gb_getLastAction()" conseguimos detectar cuándo se pulsa la barra espaciadora. Si esta es pulsada, se ejecuta un método que se encuentra en la clase "Jugador" que crea el torpedo y lo hace visible. Paralelamente, fuera del switch se ejecuta el método "movimientoDisparoJugador()" que modifica la coordenada Y del torpedo, haciendo que este suba.

Para poder implementar la eliminación de enemigos al disparar, hemos creado una "hitbox" en la clase "Nivel". Dentro de ella, el método "elimEnem()" compara las posiciones de los enemigos y el torpedo en un intervalo establecido, para así poder detectar cuando coinciden y eliminar el Sprite del enemigo haciéndolo invisible.

- SPRINT 3: (Entrada)

La entrada ha sido implementada mediante un método abstracto ("entrada ()") en la clase "Enemigos" de la que hereda cada tipo de enemigo (Zako, Goei, Commander, SuperCommander).



Al inicio de cada nivel este método se ejecuta. De forma paralela unas constantes son declaradas (Norte, Noreste, Oeste, etc.), asignando unas direcciones a los movimientos por coordenadas.

Para poder implementar un tiempo de retardo hemos creado una variable "k" como argumento del método abstracto anteriormente mencionado. Esta variable adopta un valor para cada enemigo, cuyos valores van restándose (partiendo de un valor neutro y constante, k = 0). Al volverse positiva la suma de esa k y otra variable denominada "contEntr" cuyo valor va aumentando en cada iteración comienza a moverse el Sprite. De esta manera, como cada

sprite tiene una “k” distinta comienzan a moverse con un retardo cada uno.

La ruta de los enemigos está creada dentro de la clase de cada enemigo (debido a que tratamos con un método abstracto). Cada vez que el sprite toma una nueva orientación, su imagen se ve modificada para simular la acción del movimiento.

(*La foto es un ejemplo de cómo pensamos y diseñamos el recorrido de un tipo de enemigo*)

- SPRINT 4: (Ataques)

  Para poder implementar el movimiento aleatorio, se genera un número aleatorio que, si coincide con un número establecido para cada Sprite enemigo, se ejecutará el método abstracto “mover()” mediante el cual los enemigos descienden realizando un recorrido especial para cada tipo de enemigo.

  De la misma forma, los enemigos dispararán al jugador dependiendo de otro número aleatorio con el método “disparar()”.Cuánto más alto es el nivel, mayor probabilidad hay de que los enemigos disparen.

  De forma similar a como los enemigos son eliminados ocurre con el jugador. Se compara las posiciones del jugador y los torpedos enemigos en un intervalo establecido, si estos coinciden el jugador pierde una vida. Si el jugador es alcanzado por un enemigo su vida se establece en 0 y se cierra el bucle principal y se pierde la partida.

- SPRINT 5: (Explosiones, Comandos de consola)

  Cuando la salud de un enemigo se establece en 0 se ejecuta el método “enemMuerto()” encontrado en la clase nivel que cambia la imagen de los enemigos por las imágenes de las explosiones. Su funcionamiento consiste en comprobar la imagen actual del Sprite y poner la siguiente de la animación. Esto ocurre hasta que las imágenes de las explosiones son realizadas, poniéndose el enemigo como invisible. De la misma manera ocurre con el jugador, con el método “jugadorMuerto()” de la clase “Nivel”.

  A partir del nivel 5 hemos implementado un nuevo enemigo (“SuperCommander”) que es un *Commander* modificado tanto en tamaño como en color y vida.

  Para programar los comandos de la consola hemos utilizado el mismo “switch()” utilizado anteriormente para los movimientos del jugador añadiendo nuevos casos. Entre ellos encontramos:

    - “godmode”: vuelve al jugador inmune.
    - “nivel-x”: donde la x es el nivel al que se quiere acceder.

CONCLUSIONES

Con esta práctica hemos podido darnos cuenta de la utilidad que muchos conceptos aprendidos en clase tienen debido a que hasta que no vimos su potencial lo tratábamos como algo irrelevante.

En cuanto a los problemas en la práctica hemos encontrado fallos en las imágenes proporcionadas (algunas no están rotadas) aunque no ha afectado a la realización de esta.

Debido a la falta de tiempo nos han quedado ideas por implementar y creemos que podría ser mejorable en ciertos aspectos. Entre estas ideas se encontraban:

- Añadir “power-ups”
- Mayor cantidad de comandos de consola
- Disparos alternativos (no solo los torpedos)
- Mayor cantidad de niveles
- Mejor diseño en el *final boss*
- Easter Eggs