APPLICATION OF POLYMORPHISM, INHERITING AND INTERFACES TO DEVELOP AN ARCADE GAME SPACE INVADERS USING JAVA NETBEANS

Muro Giurfa Freddy
e-mail: mur._@hotmail.com

Escuela Profesional de Ingeniería Informática
Universidad Ricardo Palma

ABSTRACT: *With this project we demonstrate that as a level three student (program curricula) programming knowledge it is possible to build an event based arcade like game using Basic concepts of inheriting, polymorphism and interfaces. We followed the professor Alexander Hristov model and we also added multiplayer capabilities.*

RESUMEN: *Con el presente proyecto se demuestra que con los conocimientos de tercer ciclo se puede desarrollar un pequeño programa de eventos totalmente funcional, en este caso un Juego, en el cual se utilizan conceptos fundamentales de la Programación orientada a Objetos como Polimorfismo, Herencia, Interfaces, etc. Se ha seguido el modelo del profesor Hristov y se han añadido ideas propias, como la implementación de otro disparo, un final al juego y lo más interesante la funcionalidad de multijugador.*

1 INTRODUCCIÓN

El Proyecto de Space Invaders es un videojuego básico y funcional hecho en java, manejado por eventos, graphics y herencia. Para su elaboración se necesitan conocimientos de Java tales como el manejo de eventos, la implementación de sonidos, la herencia, el polimorfismo. El JAR es un videojuego de 10 MB, totalmente funcional que demuestra el gran potencial y la versatilidad que tiene Java.

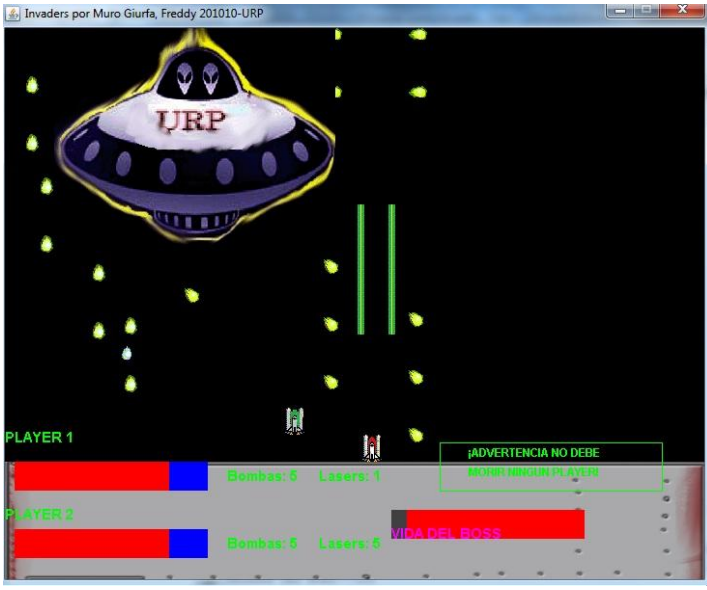
Siguiendo el modelo del profesor Hristov (ver link a su página web en referencias) se logró realizar este juego, que implementa el modelo Hristov sumado con las ideas propias del expositor, como la implementación de otro disparo, un final al juego y lo más interesante la funcionalidad de multijugador.

2 PRESENTACIÓN DEL PROBLEMA

2.1 CARACTERÍSTICAS GENERALES

Conceptos Usados:

1. Uso de Eventos.
2. Manejo de Excepciones.
3. Aplicaciones con imágenes y sonidos.
4. Herencia.
5. Polimorfismo.
6. Métodos Abstractos.
7. Interfaces.
8. Dibujo y Dibujo 2D.
9. Métodos, sobrecarga y constructores.
10. Inteligencia Artificial Básica (Guiada por Randoms).



2.2 DIBUJOS EN EL JFRAME

Para realizar los dibujos en el JFrame, necesitaremos un panel (lienzo) sobre el cual hacer el pintado, seguido de esto, necesitaremos una mención de la imagen que queremos emplear, haciendo uso de los métodos de URL y ImageIcon para ello.

2.2.1 MANEJO DE IMÁGENES

Las imágenes serán pintadas mediante un método que las pinte con una variable de posición, siendo este la velocidad, por ejemplo si ponemos velocidad 5, querrá decir que se moverá de 5 en 5 espacios en la dirección que se especifique.

2.3 PROPIEDADES DEL JFRAME

Para que el "escenario" tenga valores fijos, sea dinámico, y reciba eventos(estímulos) habremos de hacer una serie de Imports de librerías, implementación de interfaces y además trabajar con los Events de la pantalla que podamos necesitar.

2.4 ¿CÓMO SE LOGRAN LOS MOVIMIENTOS?

Como todos sabemos, para realizar animaciones en java se puede optar por usar un timer con métodos definidos en el timer task, o emplear un buffering para la carga y el pintado de las imágenes.

2.4.1 ¿QUÉ ES UN BUFFERING?

Los buffering son la captura de la imagen antes de ser borrada, que se almacena en la memoria de la maquina y se proyecta en el programa, a la hora de borrar y repintar la imagen en otro lado, el usuario sigue viendo esta para luego ver que se"mueve" hacia el repintado sin ver el molesto parpadeo.

2.5 ¿CÓMO SE DISTRIBUIRÁN LAS CLASES?

Para el desarrollo del programa, debemos de tener en claro que cosa va a hacer cada objeto que creemos, tenemos que buscar una manera en la que la ejecución del código sea de manera limpia y sobretodo versátil.

2.6 ¿CÓMO SE RELACIONAN TODOS LOS OBJETOS ENTRE SÍ?

Si queremos que el videojuego cobre sentido vamos a tener que hacer una serie de procedimientos para que 2 objetos o más, reconozcan que están en interacción y actúen de algún modo.

2.7 METODOS IMPLEMENTADOS

Para el manejo del daño de la nueva clase creada llamada "Boss" se implementó

public void checkCollisionsBoss(), que permite reconocer si un objeto de esta clase está haciendo algún tipo de colisión.

```
public void checkCollisionsBoss() {
  Rectangle bossBounds =  boss.getBounds();
  for (int i = 0; i < actors.size(); i++) {
  Actor a1 = (Actor) actors.get(i);
   Rectangle r1 = a1.getBounds();
       if (r1.intersects(bossBounds)) {
             boss.collision(a1);
             a1.collision(boss);
          }
}
```

Para terminar el juego, se implementaron 2 métodos (ganar o perder) que se emplean en el momento adecuado: **public void paintGameOver(Graphics g)** y **public void paintBossOver(Graphics g)**

```
public void paintGameOver(Graphics g) {
      g.setColor(Color.WHITE);
      g.setFont(new Font("Arial", Font.BOLD, 15));
      g.drawString("¡PERDISTE!",Stage.WIDTH/2-60,
  Stage.HEIGHT / 2);
      ac.stop();
      soundCache.playSound("cry.wav");
  }
   public void paintBossOver(Graphics g) {
      g.setColor(Color.WHITE);
      g.setFont(new Font("Arial", Font.BOLD, 15));
       g.drawString("¡GANASTE!", Stage.WIDTH/ 2 - 60,
Stage.HEIGHT / 2);
       ac.stop();
       soundCache.playSound("victory.wav");
  }
        for (int j = i + 1; j < actors.size(); j++) {
           Actor a2 = (Actor) actors.get(j);
           Rectangle r2 = a2.getBounds();
           if (r1.intersects(r2)) {
              a1.collision(a2);
              a2.collision(a1);
           }
        }
     }
  }
```

Estos métodos son invocados desde el método principal de pintado de pantalla, con una condición If(bossmuerto==true).

Además este método permitirá al programa determinar si se ha escogido el modo multijugador al inicio del juego. En un principio se quiso también poner la opción para escoger numero de enemigos y dificultad, pero al final no era nada cómodo(funcionaba, pero el juego carecía de gracia).

```
public void game() {
      usedTime = 1000;
      initWorld();
      while (isVisible() && !gameEnded) {
         long startTime = System.currentTimeMillis();
         updateWorld();
         checkCollisions();
         if (multijugador) {
            checkCollisions2();
         }
         if (existeBoss) {
            checkCollisionsBoss();
         }
```

```
        paintWorld();
        usedTime=System.currentTimeMillis()- startTime;
        try {
            Thread.sleep(SPEED);
        } catch (Exception e) {
        }
    }


    Graphics g = this.getGraphics();
    if (bossEnded) {
        paintBossOver(g);
    } else {
        paintGameOver(g);
    }
```

Así mismo, para el método de pintado de pantalla y refrescamiento de la misma, se tuvo que implementar los métodos para visualizar al nuevo jugador y al boss:

```
public void paintWorld() {
Graphics2D g = (Graphics2D)
strategy.getDrawGraphics();
        g.setColor(Color.black);
        g.fillRect(0, 0, getWidth(), getHeight());
        for (int i = 0; i < actors.size(); i++) {
            Actor m = (Actor) actors.get(i);
            m.paint(g);
        }
        player.paint(g);
        if (existeBoss) {
            boss.paint(g);
        }
        if (multijugador == true) {
            player2.paint(g);
            paintFondo(g);
            paintStatus2(g);
        } else {
            paintFondo(g);
            paintStatus(g);
        }
        if (existeBoss) {
            paintBossShields(g);
        }
        g.setColor(Color.white);
        strategy.show();
    }
```

Para poner el fondo se uso un método simple que recibe como parámetro un objeto de la clase Graphics2D :

```
public void paintFondo(Graphics2D g)
{  URL url = getClass().getClassLoader().
     getResource("res/fondo00.gif");
       try {
           BufferedImage fondo = ImageIO.read(url);
           g.drawImage(fondo, 0, 450, this);
       } catch (Exception e) { }
    }
```

Se puede apreciar que graficar los efectos de atacar al enemigo boss, es un pintado de 2 rectángulos, uno estático y otro que se va encogiendo mientras siga recibiendo un determinado estímulo , esto se controla mediante el método **public void paintBossShields.**

```
public void paintBossShields(Graphics2D g) {
        int alturaboss = 500;
        g.setPaint(Color.red);
        g.fillRect(400, alturaboss, boss.MAX_SHIELDS, 30);
        if (boss.getShields() >0) {
            g.setPaint(Color.darkGray);
        }
        g.fillRect(400, alturaboss, boss.MAX_SHIELDS –
                                        boss.getShields(), 30);
        g.setFont(new Font("Arial", Font.BOLD, 15));
        g.setPaint(Color.MAGENTA);
        g.drawString("VIDA DEL BOSS", 400,
                                                alturaboss + 30);
    }
```

El método Main, desde donde se ejecutan los métodos en un bucle mientras no se pierda o se gane:

```
public static void Ejecutar() {
        numJugador = Integer.parseInt(
JOptionPane.showInputDialog("NUM JUGADORES?1 o 2"));
        if (numJugador == 2) {
            multijugador = true;
JOptionPane.showMessageDialog(null, "Jugador 1 se mueve
con las flechas y dispara con 1,2,3(numpad).\n Jugador2 se
mueve con AWSD y dispara con y,u,i");
        }
    else {
         JOptionPane.showMessageDialog(null, "Jugador1
        se mueve con las flechas y dispara con 1,2,3(numpad)");
        }
        dificultad = 3;
        Invaders inv = new Invaders();
        inv.game();
    }
    public static void main(String[] args) {
        Ejecutar();
    }
```

2.8 MANEJO DE EVENTOS

Para que el jugador pueda controlar su nave, es necesario que la ventana reciba y sepa procesar los estímulos del teclado. Para ello es necesario implementar los keyListener para usar el KeyPressed de esta forma se logra un adecuado manejo de eventos teclado.

2.8.1 MOVIMIENTO DE LOS JUGADORES.

Los jugadores se moverán gracias al estimulo del teclado, que será capturado y evaluado para pintarse en una dirección en concreto. Esto se logra mediante métodos en la clase del jugador.

2.9 INTELIGENCIA ARTIFICIAL BÁSICA

Para que los enemigos se muevan, y disparen (en resumen actúen) de manera automática, necesitaremos programar una ligera inteligencia artificial capaz de reconocer el movimiento a seguir, los disparos y cuando “desaparecer”.

2.10 CICLO DEL JUEGO

Para que el proyecto funcione adecuadamente precisaremos que siga un “ciclo” constante de refresco de pantalla y actualización de eventos, para esto haremos uso de diversos métodos que se encargaran de manera automática el pintado necesario. Así mismo es esencial que se espere un cierto intervalo de tiempo entre ciclo y ciclo para que el usuario pueda apreciar de manera ideal los cambios que ocurren dentro del juego.

2.11 TERMINO DEL JUEGO

Para que la aplicación termine, se debe ganar o perder, y para que el programa reconozca estos eventos será necesario trabajar con variables tipo flag o booleanas, las cuales indicarán si el juego ha acabado.

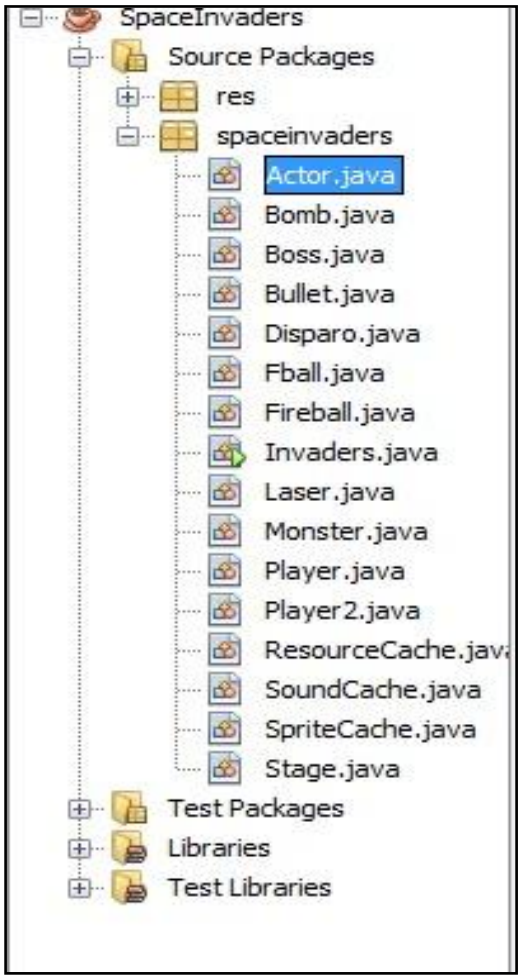
Luego de declarado el final del juego se pondrán los avisos de “victoria” o “derrota” (seguido de los respectivos sonidos) según sea el caso.

3 DESCRIPCIÓN DE LA SOLUCIÓN

En el siguiente punto se aclarara de manera general varios puntos del funcionamiento del programa. Explicaciones de cómo funciona determinado código “esencial” que cabe resaltar.

3.1 CLASES NECESARIAS

Se usaron en total 16 clases para el adecuado manejo del proyecto.



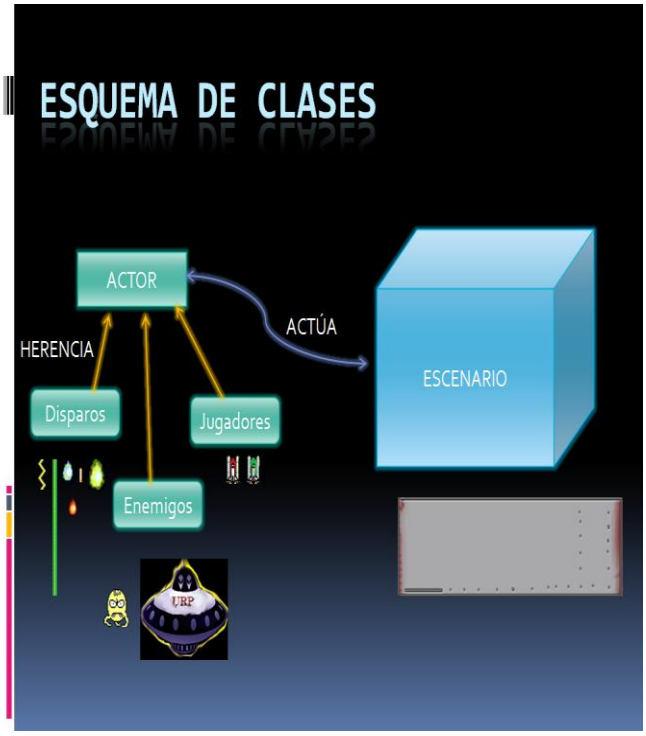
3.2 PINTADO DE ACTORES

Para ello fue necesario el siguiente codigo:

```
URL url =
getClass().getClassLoader().getResource("res/imagen.gi
f");
BufferedImage img =  ImageIO.read(url);
      g.drawImage(img, X, Y,this);
```

Siendo ***url*** la dirección de referencia para las imágenes. El BufferedImage sirve para pintar una imagen del tipo buffered, empleando para el pincel “g” las posiciones x, y las coordenadas donde se ubicará el dibujo. El “this” es el que recibirá el dibujo. En nuestro caso, el JPanel.

3.3 ESQUEMA DE CLASES



3.4 PINTADO DE ACTORES

Para ello fue necesario el siguiente codigo:

```
URL url =
getClass().getClassLoader().getResource("res/imagen.gi
f");
BufferedImage img =  ImageIO.read(url);
        g.drawImage(img, X, Y,this);
```

Siendo ***url*** la dirección de referencia para las imágenes. El BufferedImage sirve para pintar una imagen del tipo buffered, empleando para el pincel “g” las posiciones x, y las coordenadas donde se ubicará el dibujo. El “this” es el que recibirá el dibujo. En nuestro caso, el JPanel.

3.5 IMPLEMENTAR SONIDOS

El método más sencillo y funcional:

```
URL url=this.getClass().getResource(“sonido.wav”);
AudioClip ac=Applet.newAudioClip(url);
 ac.play();
```

Como se aprecia, el AudioClip es creado mediante el uso de un Applet, y se llena el parámetro con una url que hace referencia al sonido que se desea reproducir. Solo se soporta sonidos con extensión .au, .wav entre otros. En el proyecto se han utilizado .wav para los sonidos, y .gif para las imágenes.

3.6 BUCLE DEL JUEGO

El juego es una serie de pintados y de actualización de pantalla, que reacciona ante los eventos y responde de manera lógica. Esto se logra teniendo el siguiente “bucle de juego”:

- Actualizar el "estado del mundo" o el "escenario”
- Si es preciso, refrescar la pantalla
- Esperar un intervalo de tiempo
- Volver al punto 1

3.7 MANEJO DE MOVIMIENTO ENEMIGOS

Los enemigos siguen una secuencia de desplazamiento de izquierda a derecha, para que sea dentro de los marcos del frame, se establece la condición:

Si la posición X del actor es igual a 0, o si esta es mayor al ancho del Frame, entonces la velocidad(o sea desplazamiento del dibujo) se invierte. If(X==0 || X>Frame.Width) Vx=-Vx;

3.8 MANEJO DE MOVIMIENTO DEL JUGADOR

El jugador se desplaza mediante estímulos que son capturados en el frame y leídos por métodos.

Se mueve mediante los eventos de key pressed, por ejemplo: KeyEvent.VK_UP será el estimulo de la tecla arriba, y esto generará que su posición en el eje Y disminuya.

Si su posición X, Y es mayor a Frame.Widht ó Frame.Height entonces, ó es 0 , entonces Vx ó Vy se ponen a 0, para que no sigan en movimiento.

3.9 COLISIONES

3.9.1 ¿QUÉ SON?

La colisión es la verificación si 2 instancias se encuentran en intersección. Cuando 2 objetos entran en contacto o interacción de alguna manera.

3.9.2 ¿CÓMO FUNCIONA?

Se tiene un método para que cada Actor que se crea tenga (de manera “invisible”) un rectángulo que lo encierra. Si este rectángulo intercepta el de otro objeto específico (Otro actor), provocará colisión

.

El rectángulo invisible tiene el método (que devolverá un booleano) a instanceof b. Para determinar si a esta en colisión con b

.

3.10 DISPAROS

3.10.1 DISPAROS ENEMIGOS

Están dados por una instrucción de aleatoriedad. Se usa un Math.random, y una variable int frecuencia=0.01, y si frecuencia> Math.random(); entonces que dispare.

Mientras más se acerque a 1.00 la frecuencia, mas disparos por "actualización de pantalla" realizará el enemigo.

Los disparos tienen además, un comportamiento que hace que desaparezcan cuando su posición es mayor a la del Frame.Height, o cuando colisiona con el jugador.

3.10.2 DISPAROS DEL JUGADOR

El comportamiento de los disparos del jugador es:

Ser borrados si colisionan con algún enemigo. O si su posición Y es menor que 0. Cada Disparo hecho disminuye la vida del Boss en 3 puntos.

- **Disparo Normal:** Se pinta la bala desde la posición Y de la nave, y su posición va disminuyendo.
- **Láser:** Tiene la misma lógica del disparo normal, solo que se pintan 3 láseres a la vez.
- **Bombas:** Son disparos que salen en 8 direcciones, tienen Velocidad X y Velocidad en Y.

3.11 LÓGICA DEL JUEGO

Puede ser jugado en mono usuario o multijugador. El objetivo es derrotar a todos los enemigos, que buscarán eliminar al jugador. Después de eliminar a los enemigos normales aparecerá el Boss que supone un reto a pasar para poder finalizar el juego.

El número de enemigos es de tipo fijo (siempre 15), y una vez eliminados aparecerá el enemigo final. Asi mismo, el juego fue diseñado para que sea mucho más fácil en modo cooperativo (2 jugadores).

4 RESULTADOS

Como resultado una mención de manera general, seria resaltar:

- Un videojuego hecho en java.
- Una aplicación con una lógica orientada a objetos.
- Pintados automáticos, que siguen una "actualización" de pantalla.
- Una "actuación limpia" por parte del código, y performance de las animaciones, así como de la secuencia del juego(personaje->eliminar enemigos->enfrentar reto final->fin del juego).
- Inteligencia Artificial Básica guiada por una serie de números aleatorios y comparaciones de "mayor menor".
- Lógica del manejo de eventos y estímulos, como capturarlos y ordenarles que ejecuten acciones (comandos).
- Una adecuada Jerarquía de Clases, que convenientemente ahorran código, y están "especializadas" según la necesidad.

5 CONCLUSIONES

El presente proyecto ha servido al alumno de sobremanera. Ayudando a entender el significado del lenguaje de programación orientado a objetos, aplicando el paradigma de la programación "todo es un objeto".

Así mismo, cabe resaltar que el proyecto necesito de muchísimos métodos para el adecuado funcionamiento y clases que hereden propiedades comunes (como en el esquema citado más arriba).

Por más difícil que pueda parecer un trabajo, se hará ameno cuanto antes se tenga claro que nada es imposible, y que con la ayuda de los docentes y la propia investigación lo complejo se vuelve un reto, y proporciona satisfacción y entretenimiento.

También cabe resaltar que Java, como lenguaje versátil que es, nos da las herramientas para un adecuado manejo de eventos y multimedia, que nos permite el día de hoy apreciar muchas aplicaciones de todo tipo en la vida cotidiana.

Dando Cifras: Para el desarrollo del proyecto, fueron necesarias 16 clases, 84 métodos (sin contar setters ni getters), más de 30 ítems de multimedia (sonidos e imágenes) y muchas horas de trabajo.

Finalmente, para concluir el informe, es necesario hacer un llamado para motivar la investigación en la educación, porque tarde o temprano será la luz que guíe la educación en el Perú, ya que en estos tiempos no se presta la importancia necesaria.

REFERENCIAS

[1] *Alexander Hristov*, *Videojuego SpaceInvaders JAVA* http://www.ahristov.com/proyectos.jsp

[2] JavaTM 2 Platform, Standard Edition, *Oracle* http://download.oracle.com/javase/1.4.2/docs/api/overview-summary.html