Módulo Programación.

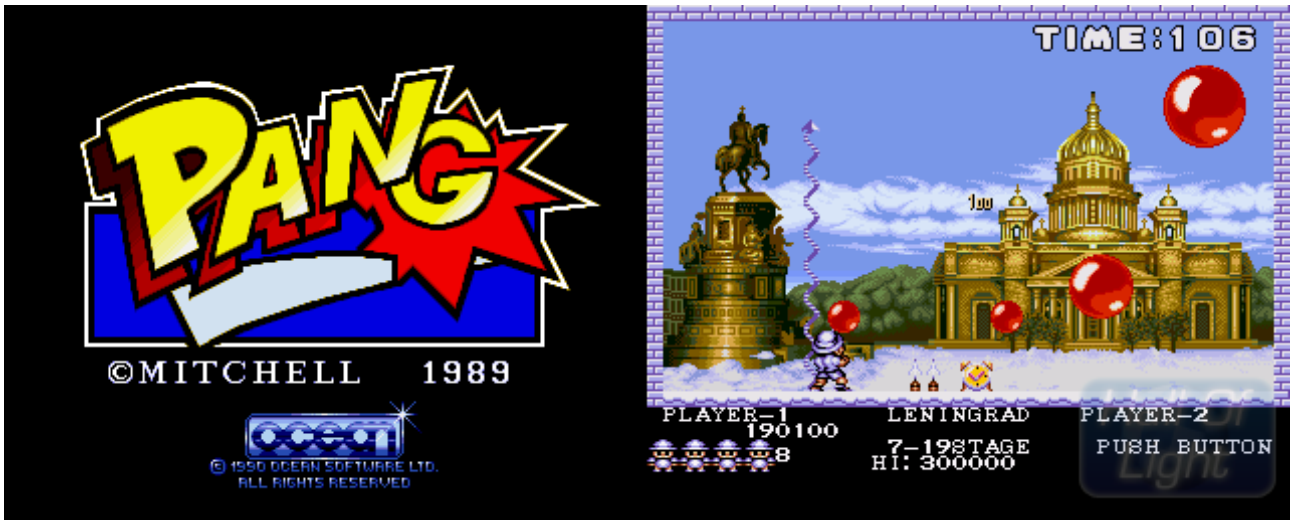
1º DAW.

PRÁCTICA: POO Avanzada y Excepciones.
UNIDAD DE TRABAJO 5.
Profesor: Pedro Antonio Santiago Santiago.

1. Introducción.

En esta práctica se trabaja con otro de los principios básicos de la programación orientada a objetos. La herencia permite la reutilización de código, la extensibilidad de este y en menor medida código con menos errores .

Se crea una versión simplificada del famoso juego “Pang” o “BusterBros” en la que se crean y utilizan diferentes clases y objetos que hacen uso de la herencia.



Además en código generado habrá de controlar las posibles excepciones que se produzca, así como lanzar excepciones en los lugares necesarios, por ejemplo en selección alternativa en el que se de un caso que no es posible.

2. Materiales.

Ordenador con JDK 11 instalado.

Netbeans 14 o superior.

Proyecto plantilla con Maven en GitHub: https://github.com/pass1enator/BusterBrosTemplate.git.

3. Desarrollo de la práctica.

A partir del código que se facilita crear el juego Pang.

3.1. Descripción del juego.

Las características del juego extraídas de la Wikipedia y adaptadas a la práctica, siendo estas:

Cada etapa o nivel contiene una distribución diferente de bloques, algunos bloques desaparecen (los que son cristales) después de ser disparados, otros no se pueden romper (los que no).

Las pantallas empiezan con diferente número y tamaño de bolas. La bola se va dividiendo en los primeros tres disparos; después del cuarto la bola es lo más pequeña posible y al dispararle desaparece.

Las armas del jugador pueden ser:

- Un arpón sencillo. Con el que se inicia cada nivel.
- Un doble arpón que permite hacer dos disparos a la vez. Especialmente útil cuando hay muchas bolas pequeñas.
- Un gancho de agarre que se pega al techo o bloque durante un breve período. Es especialmente útil con techos bajos y espacios estrechos.
- Una pistola de alto calibre que funciona como un ametralladora que permite rápidos disparos. Especialmente útil con bolas grandes.

Se podrá cambiar de arma pulsando la tecla W. No hay límite de munición en cualquier arma.

Si un jugador toca una bola de cualquier tamaño, el jugador muere (se le resta una vida y se deber reiniciar el nivel de quedar vidas)

Los jugadores empiezan con 3 vidas.

La etapa o nivel finaliza cuando todas las bolas han sido eliminadas por completo.

El juego finaliza cuando todas las etapas han sido superadas con éxito.

Extras:

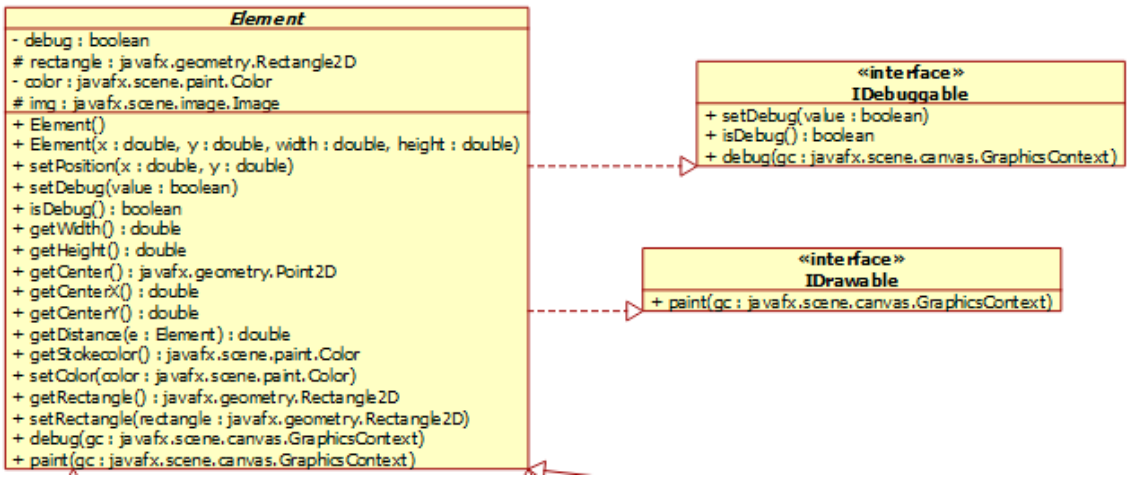
Existen una serie de bonificaciones que aparecen al romper un bloque o al dividir una bola:

- Una Fuerza de campo. Protege al jugador del impacto de una bola.
- Un reloj de arena. Disminuye la velocidad de caída de las bolas.
- Un reloj. Para las bolas por un corto tiempo.
- Una vida extra.
- Dinamita. Explota todas las bolas que hay en bolitas diminutas.

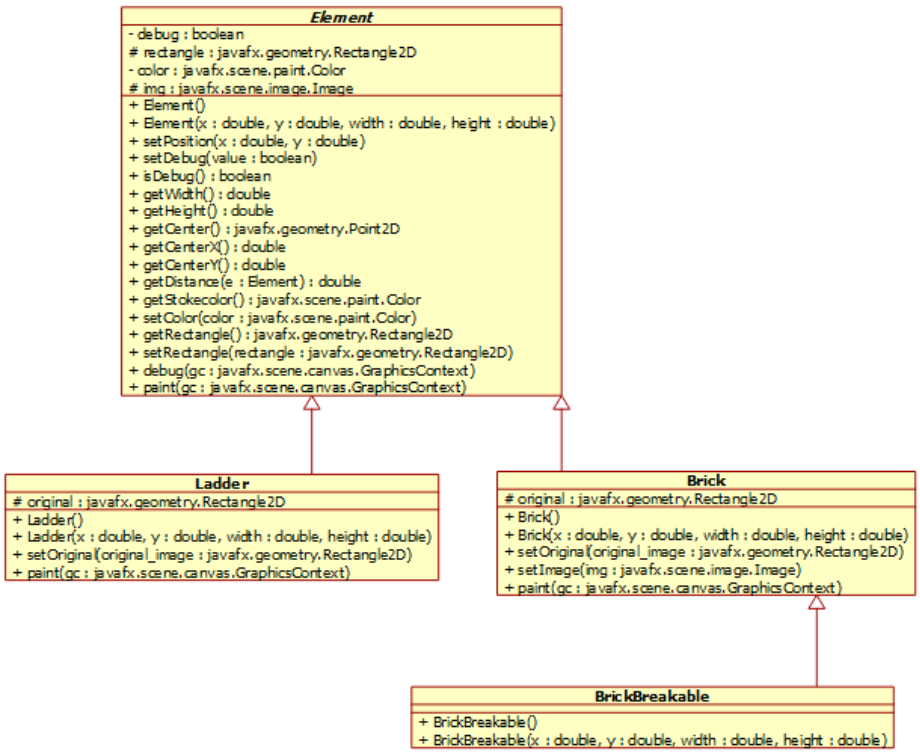
3.2. Propuesta de clases.

El juego básico se compone de bloques, bloques rompibles, escaleras, jugador y bolas.

Todos los elementos anteriores poseen un ancho, alto y posición (x e y). Ya que todos poseen estos atributos, se define la clase abstracta **Element**, que implementa las interfaces IDebuggable para la depuración (por ejemplo dibujar área, coordenadas centro, extremos…) y la interfaz Idrawable para que se pueda dibujar en el lienzo. El diagrama de clases es el siguiente:

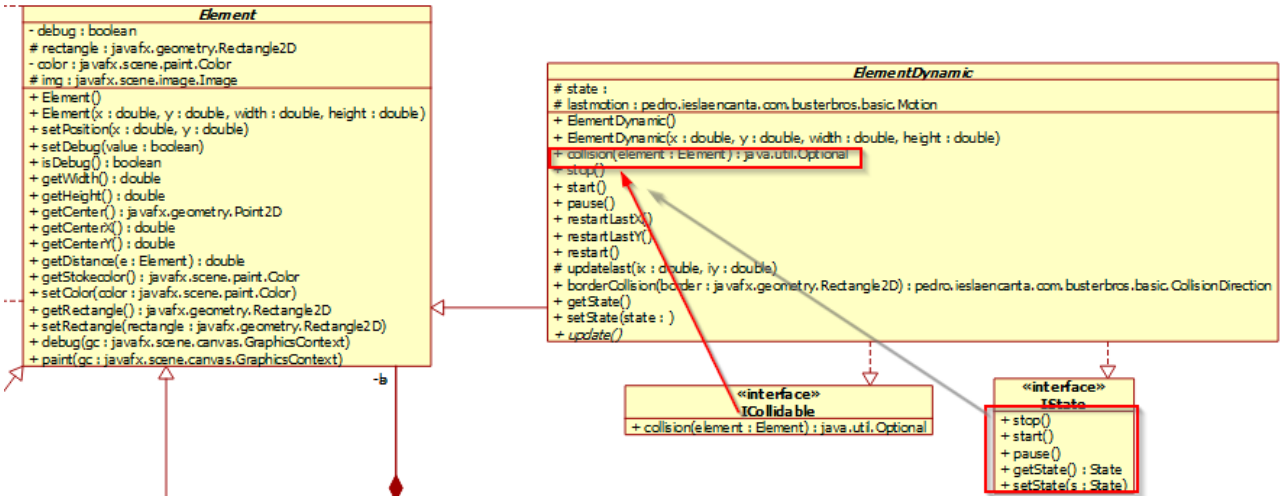


De esta clase hereda de forma directa las escaleras (**Ladder**), y los bloques, de estos últimos se tienen los fijos (**Blick)** y los rompibles (**BrickBreakable**):



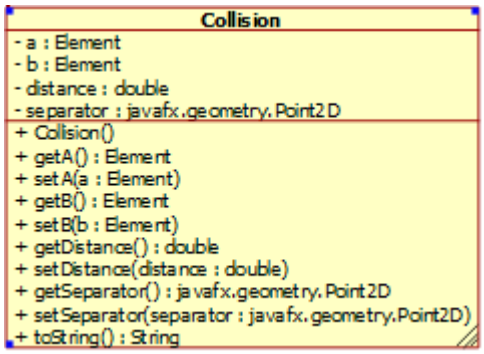
Existen también elementos que crecen o se mueven, estableciendo la clase abstracta ElementDynamic, cuya principal característica es la de que implementa la detección de

colisiones con objetos de tipo Element, con la interfaz ICollidable, además estos elementos dinámicos posee un estado e implementan la interfaz IState.



Existen multitud de forma de detectar colisiones, se ha definido uno **aunque el alumnado puede establecer el que considere más oportuno, siempre que funcione.**

El método “collision (Element) devuelve un objeto de tipo Optional, utilizado para la gestión de los nulos, es una especie de “bolsa” en la que se introduce un objeto, pudiendo preguntar si la bolsa está vacia “isEmpty()” o llena “isPresent()”.

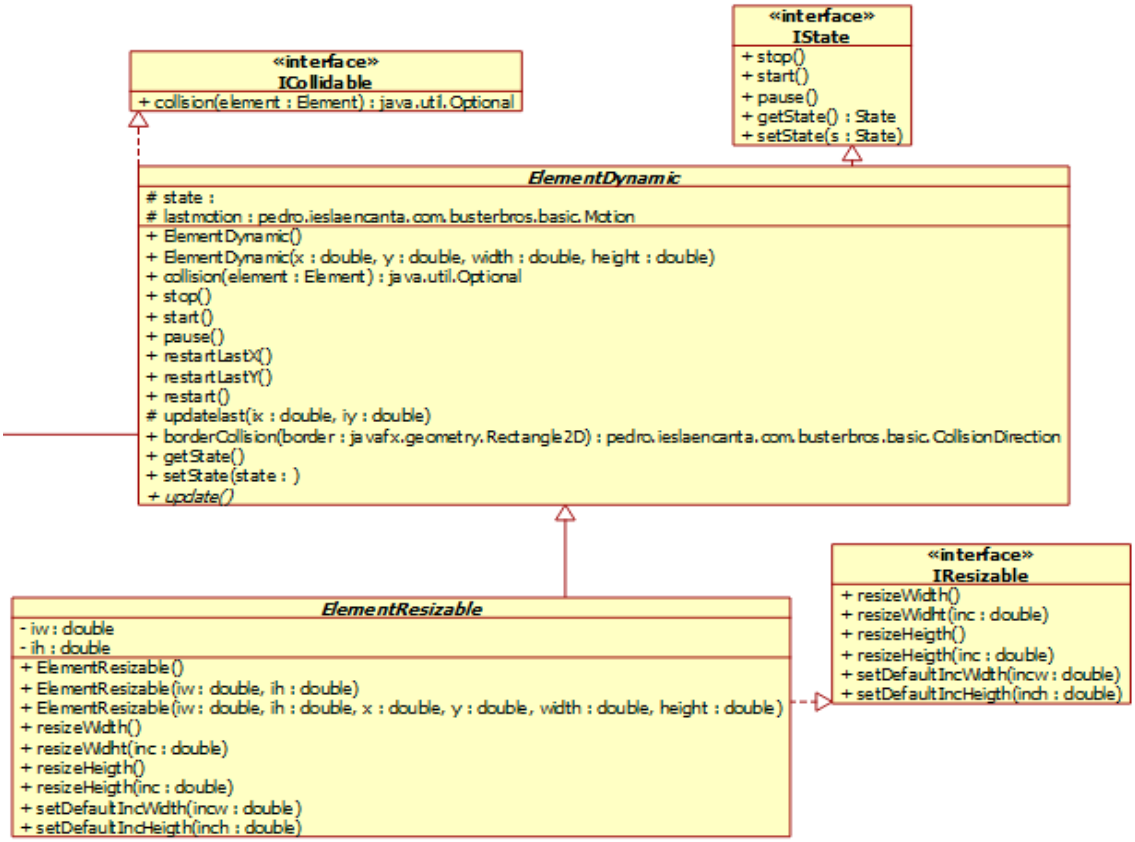


En “Optional” se almacena un objeto de tipo “Collisión” que tiene:

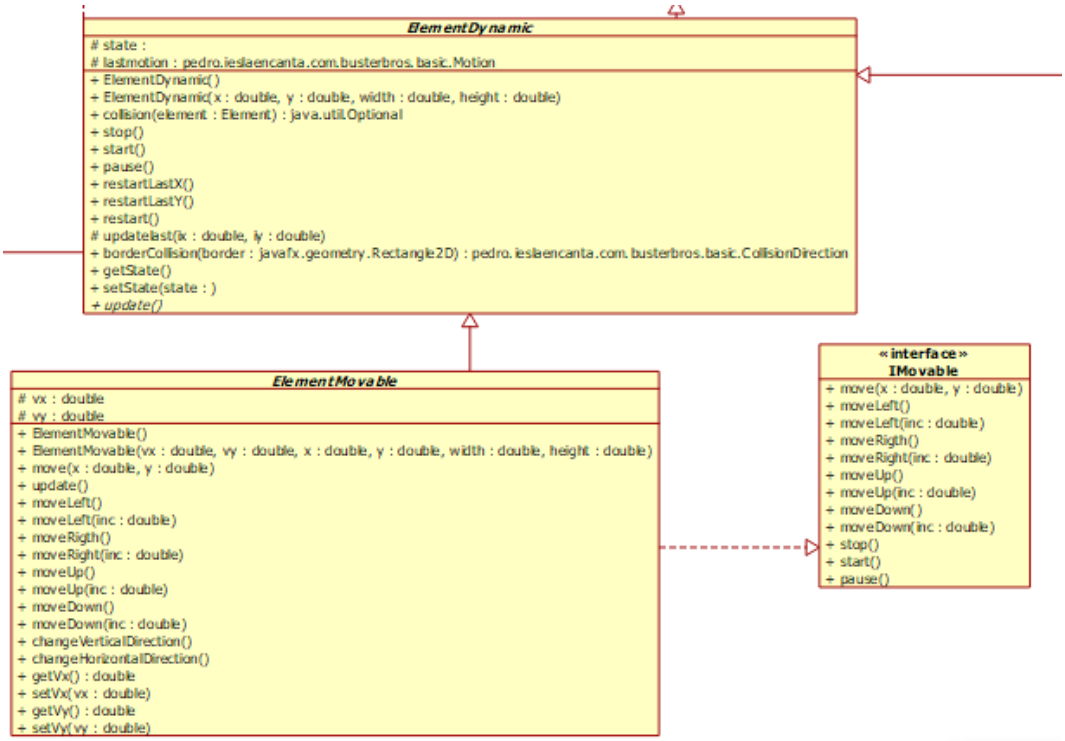
- a: Objeto de tipo “Element” que llama a “collision”
- b: Objeto de tipo “Element” contra el que colisiona.
- Distance: Distancia entre los centros de los dos objetos.
- Separator: Objeto de tipo Point2D que indica la separación en el ejex y eje y que ha de realizar el objeto **a** para dejar de colisionar con **b**.

De la clase ElementDynamic heredan 2 clases, “**ElementResizable**” y “**ElemenMovable**”, ambas abstractas.

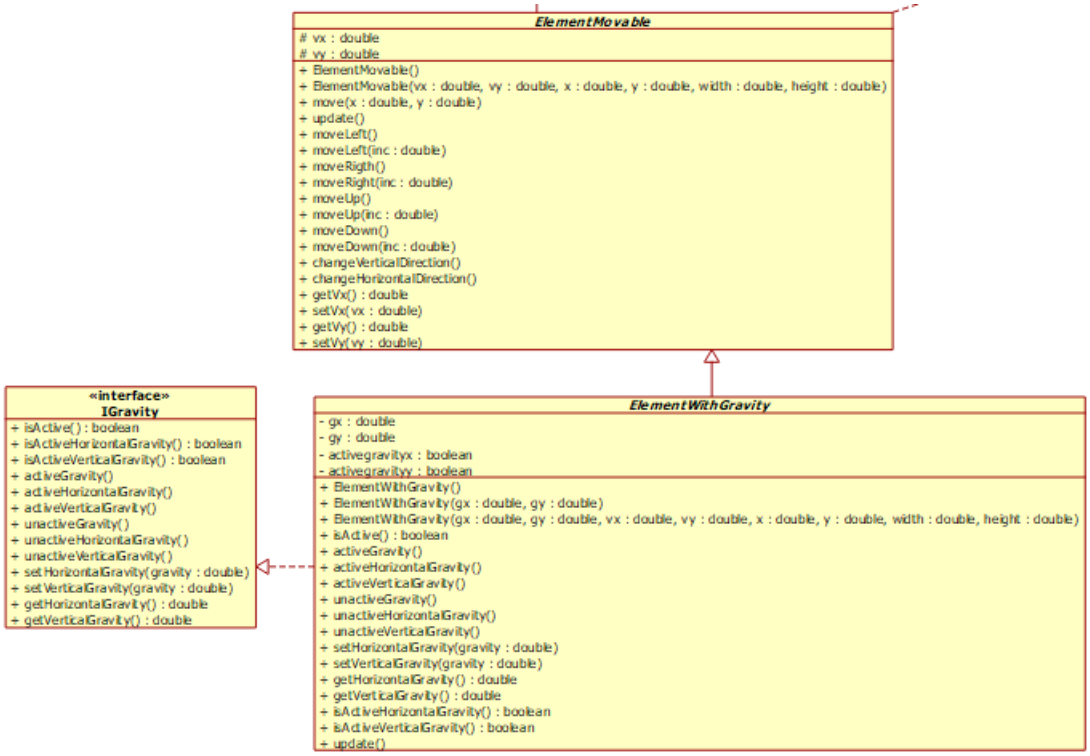
La primera permite cambiar el tamaño de un elemento, por ejemplo los arpones, implementando la interfaz "**IResizable**".



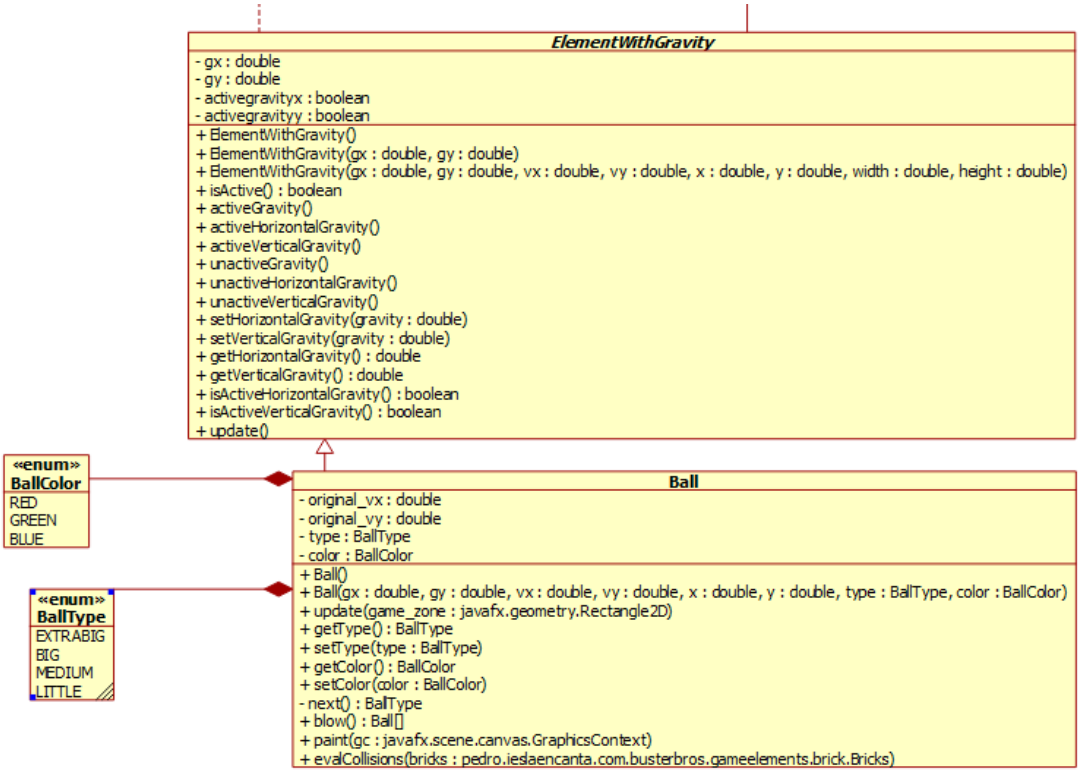
La segunda se define para elementos que se puede mover por la pantalla, como el jugador o las bolas.



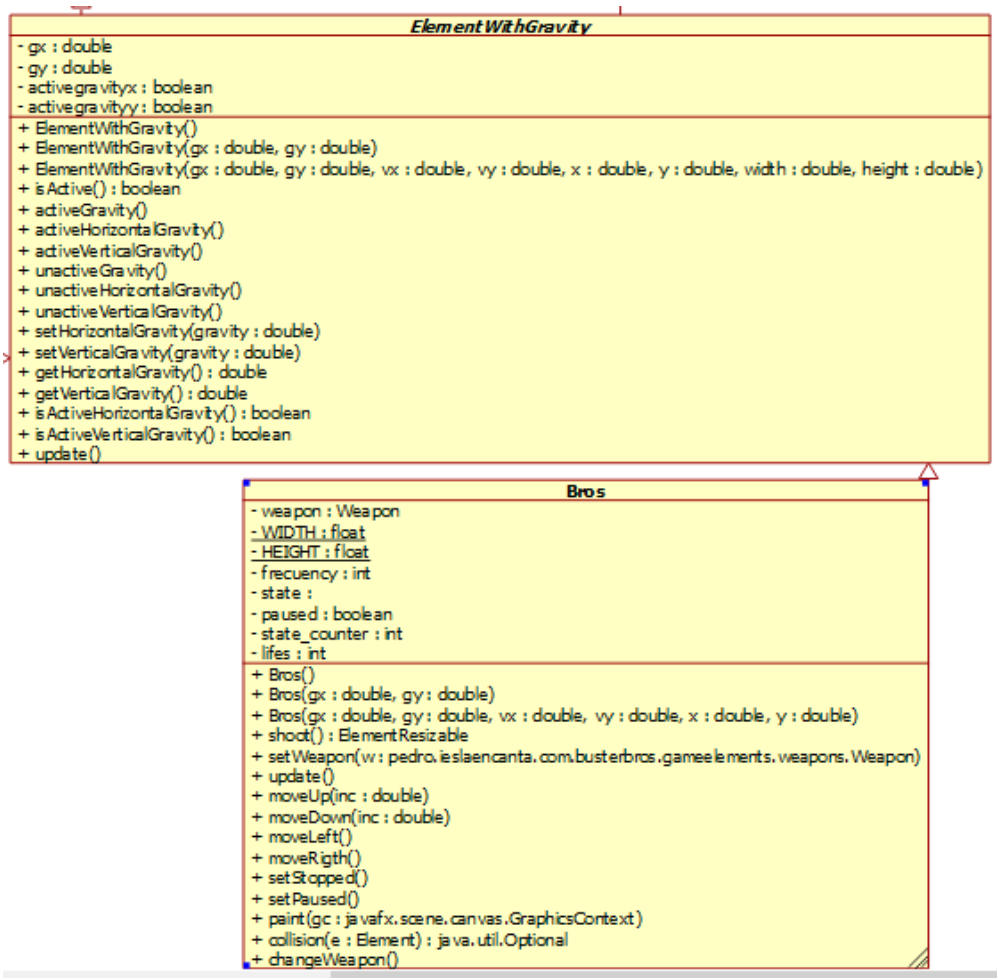
Tanto la bola como el jugador se ven afectados por la gravedad, estableciendo una nueva clase abstracta "**ElementWithGravity**" para aquellos elementos a los que les afecta la gravedad, esta clase abstracta implementa la interfaz **"IGravity"**.



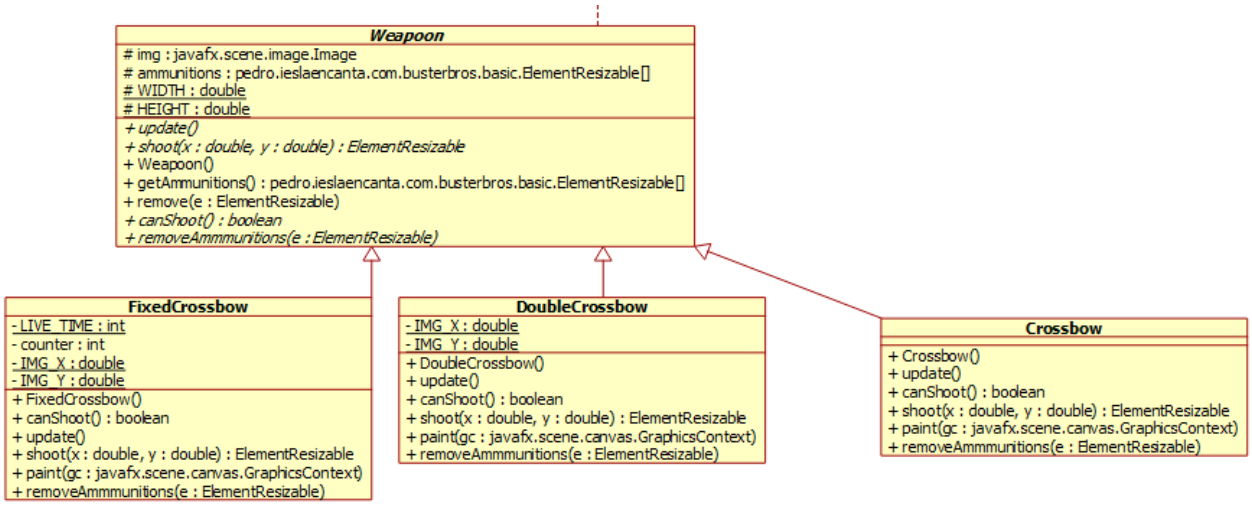
Sobre la clase abstracta "**ElementWithGravity"** se define la clase **"Ball"**, esta ya no abstracta. Esta clase a su vez se apoya en dos enumerados para el color (coordenadas imagen) y el tamaño.



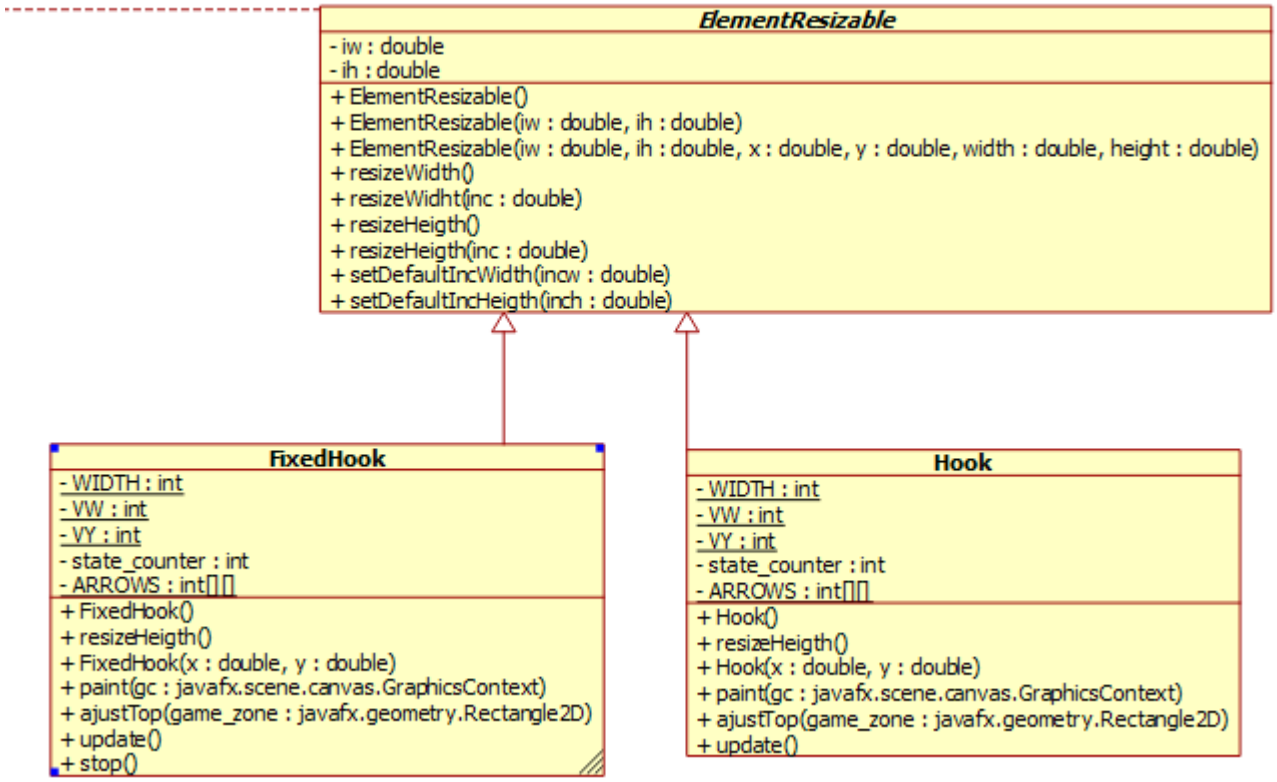
La clase que representa al jugador se llama "**Bros**" y hereda de "**ElementWithGravity**":



En cuanto a las armas, el jugador "**Bros**" posee una atributo de tipo "**Weapon**" heredando cada una de los cuatro tipo de armas de esta e implementando los métodos abstractos de la clase Weapon.



Estas "armas" poseen "municiones" de diferente tipo: balas, arpones y arpones fijos, los arpones y arpones fijos heredan de de la clase abstracta "**ElementResizable**" y las balas de "**ElementMovable**":



Todas estas clases y elementos se integran en el tablero "**Board**", que se compone entre otros atributos de:

- Jugador.
- Conjunto de bloques, tanto fijos como rompibles.
- Conjunto de escaleras.
- Conjunto de bolas.

Por ejemplo para gestionar las bolas se tiene una clase **"Balls"** con el siguiente código:

```
public class Balls {
    private Ball[] elements;
    protected int size;

    public Balls(int size) {
        this.size = size;
        this.elements = new Ball[size];
    }

    public int getSize() {
        return size;
    }
```

```
public Ball getBall(int i) {
    Ball b = null;
    if (i >= 0 && i < getSize() && this.elements[i] != null) {
        b = this.elements[i];
    }
    return b;
}

public int addBall(Ball b) {
    int index = -1;
    for (int i = 0; i < this.elements.length && index == -1; i++) {
        if (this.elements[i] == null) {
            this.elements[i] = b;
            index = i;
        }
    }
    return index;
}

public void removeBall(Ball b) {
    boolean borrado = false;
    for (int i = 0; i < this.elements.length && !borrado; i++) {
        if (b == this.elements[i]) {
            borrado = true;
            this.elements[i] = null;
        }
    }
}

public int getNumberOfElements() {
    int counter = 0;
    for (int i = 0; i < this.elements.length; i++) {
        if (this.elements[i] != null) {
            counter++;
        }
    }
    return counter;
}

public boolean isEmpty() {
    return (this.getNumberOfElements() <= 0);
}
```

```
}
```

En el tablero se han de definir la mayor parte de los métodos del juego, por ejemplo:

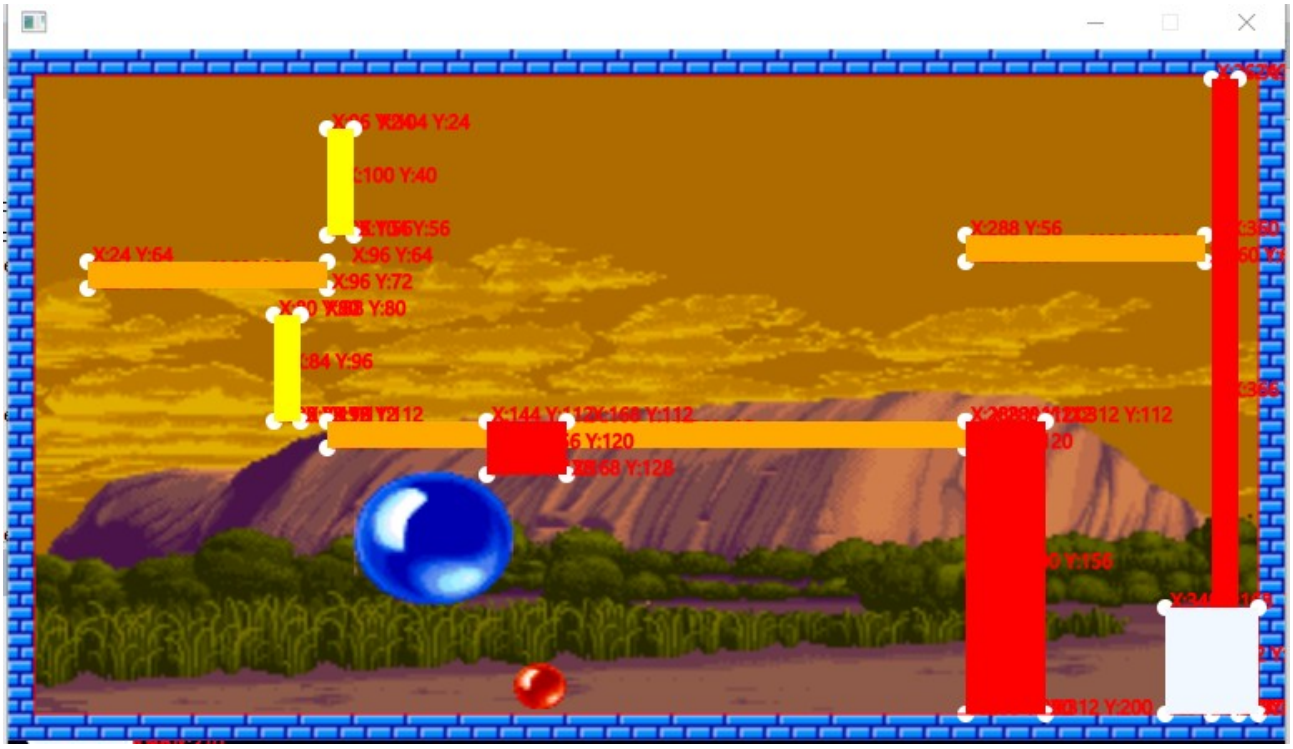
- Pasar al siguiente nivel en caso de no quedas bolas.
- Detectar las colisiones de las bolas y cambiar las direcciones.
- Detectar las colisiones del jugador con los bordes.
- Detectar la colisión de las pelotas con los bloques y actuar en consecuencia.
- Detectar la colisión del jugador con las pelotas y realizar las acciones oportunas.
- Detectar la colisión de las balas con los bloques o los bordes y establecer el comportamiento en función del tipo de bala.
- Detectar la colisión de las balas con las pelotas y explotar o eliminar.
- Gestionar usando las colisiones el movimiento del jugador por los bloques y las escaleras.

Un ejemplo de colisión entre el jugador y algún bloque, de forma que no permita atravesar el bloque, o desactivar la gravedad en caso de estar en caida:

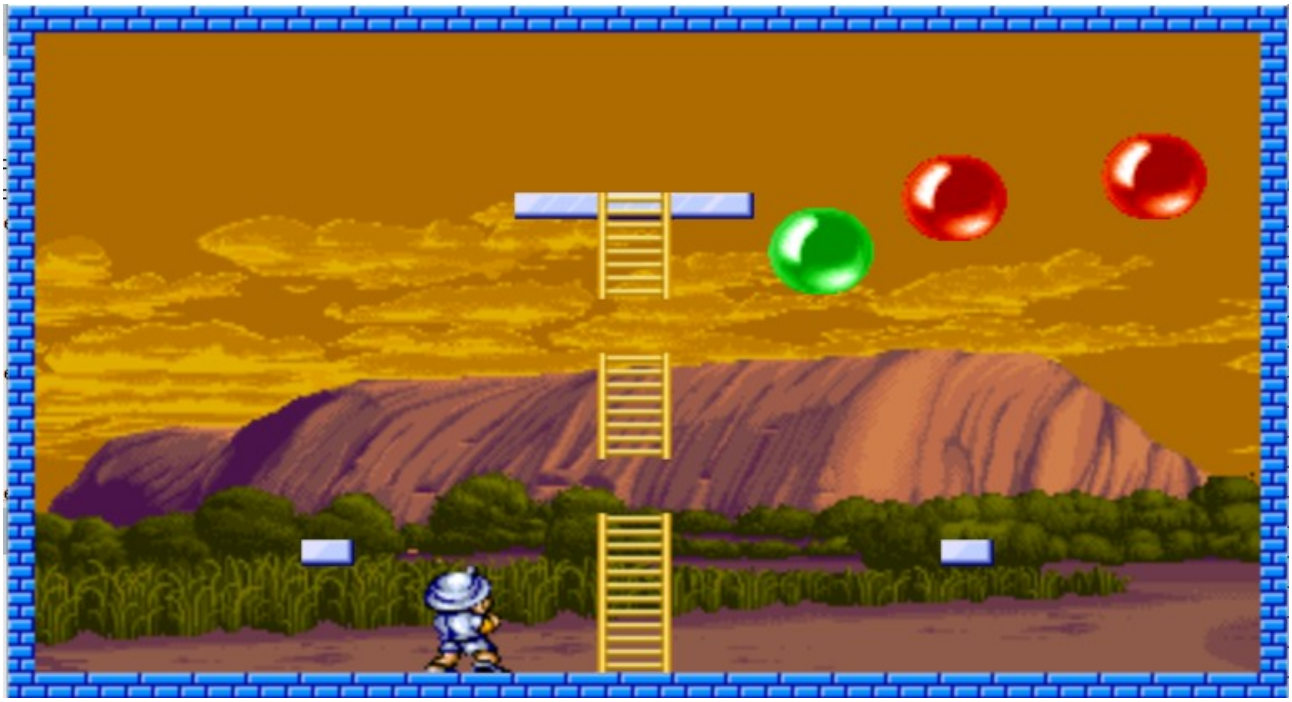
```
  public Optional<Collision> collisionWithBricks() {
        Optional<Collision> oc = Optional.empty();
        for (int i = 0; i < this.bricks.getSize() && oc.isEmpty(); i++) {
            if (this.bricks.get(i) != null) {
                oc = this.bros.collision(this.bricks.get(i));
                if (oc.isPresent()) {
                    //colision superior e inferior
                    if (oc.get().getSeparator().getY() > 0 ||
oc.get().getSeparator().getY() < 0 && !this.isInLadder()) {
                        //se para la gravedad, se actualiza la posición para el
borde y se quita la velocidad de y
                        if (oc.get().getSeparator().getY() < 0) {
                            this.bros.unactiveGravity();
                            this.bros.moveUp((oc.get().getSeparator().getY()) * -
1);
                            this.bros.setVy(0);
                        }
                    } //colisiona lateralmente
                    else if (oc.get().getSeparator().getX() > 0 ||
oc.get().getSeparator().getX() < 0 && !this.isInLadder()) {
                        this.bros.move(oc.get().getSeparator().getX(), 0);
                    }
                }
```

```
            }
        }
        return oc;
    }
```

Para las fases iniciales se recomienda no centrarse en los “spriters” sino dibujar simplemente cuadrados de diferentes colores e ir añadiendo clases y métodos.



Ir transformado los rectángulos en “spriters”:



3.3. Código inicio.

Se proporciona un conjunto de clases y objetos para facilitar el desarrollo de la aplicación.

Se tiene la clase **"Game"** que posee el "tablero" y destaca por posee métodos estáticos para el manejo de un contador que se incrementa en cada ciclo de reloj. Este contador es accesible y útil en las clases finales, por ejemplo, si se establece una frecuencia de 50 Hz, el contador se incrementa 50 veces por segundo, **se puede usar entre otras funciones para realizar la animaciones, al andar el jugador cada 3 ciclos se cambia el "spriter".**

Cada vez que se pasa de nivel es conveniente reiniciar el contador para impedir un desbordamiento.

```
public static void reset_counter() {
        synchronized (lock) {
            counter = 0;
        }
    }
    public static long getCounter() {
        synchronized (lock) {
            return counter;
        }
    }
```

También se facilita de **forma parcial** la clase Level, que representa cada uno de los niveles del juego, se le proporciona:

- La coordenada x de la imagen foreground.png.
- La coordenada y
- La imagen de la que obtiene los bloques y las escaleras.
- La imagen de fondo del nivel.
- El tiempo del nivel.

Se han de añadir a la clase nivel métodos para añadir y obtener las bolas del nivel, en el ejemplo, addBall (este método está vació)

```
this.levels[0] = new Level();
this.levels[0].setX(8);
this.levels[0].setY(y);
this.levels[0].setImagename("bricks");
this.levels[0].setBackgroundname("backgrounds");
this.levels[0].setSoundName("fondo");
this.levels[0].setTime(30);
this.levels[0].addBall(0,
                    0.05,
                    //vx
                    1,
```

```
                    //vy
                    -4.75,
                    //x
                    this.game_zone.getMaxX() / 2 - 70,
                    //y
                    this.game_zone.getMinY() + 21,
                    //w
                    BallType.EXTRABIG,
                    //h
                    BallColor.BLUE);
this.levels[0].addBall(0, 0.05,1,-2.75,this.game_zone.getMaxX() / 2 + 70,
this.game_zone.getMinY() + 21,BallType.MEDIUM,BallColor.RED);
```

Esta clase analiza la imagen y crea un conjunto de pares de tipo de elemento **"ElementType"** y rectángulos con las coordenadas en la imagen:

```
 public enum ElementType {
        FIXED,
        BREAKABLE,
        LADDER
    }
```

Sepuede recuperar con el método getFigures(), de la clase "**Level**", un ejemplo de uso en "**Board**":

```
private void createElementsLevel() {
        this.bricks = new Bricks(100);
        this.ladders = new Ladders(20);
      Pair<Level.ElementType, Rectangle2D>[] fi =
this.levels[this.actual_level].getFigures();
        for (int i = 0; i < fi.length; i++) {
            switch (fi[i].getKey()) {
                case FIXED:
                    break;
                case BREAKABLE:
                    break;
                case LADDER:
                    break;
            }
        }
    }
```

3.4. Empezando el proyecto.

Se han de ir implementando poco a poco las diferentes clases, siendo la más básica la clase "**Element**".

Esta implementa 2 interfaces básicas: "**IDebuggable**" e "**IDrawable**":

```
public interface IDebuggable {
    public void setDebug(boolean value);
    public boolean isDebug();
    public void debug(GraphicsContext gc);
}
```

```
public interface IDrawable {
    public void paint(GraphicsContext gc);
}
```

Se opta por usar dentro de "**Element**" un atributo de tipo **"Rectangle2D"** para representar la posición y las dimensiones, además se añaden diferentes métodos como calcular el centro de la figura:

```
public abstract class Element implements IDebuggable, IDrawable {

    private boolean debug;
    protected Rectangle2D rectangle;
    private Color color;
    protected Image img;

    public Element() {

        this.rectangle = Rectangle2D.EMPTY;
    }

    public Element(double x, double y, double width, double height) {
        this.rectangle = new Rectangle2D(x, y, width, height);
    }

    public void setPosition(double x, double y) {
        this.rectangle = new Rectangle2D(x, y, this.rectangle.getWidth(),
this.rectangle.getHeight());
    }

    @Override
    public void setDebug(boolean value) {
        this.debug = value;
```

```
    }

    @Override
    public boolean isDebug() {

        return this.debug;
    }

    public double getWidth() {
        return this.getRectangle().getWidth();
    }

    public double getHeight() {
        return this.getRectangle().getHeight();
    }

    public Point2D getCenter() {
        return new Point2D(this.getCenterX(), this.getCenterY());
    }

    public double getCenterX() {
        return this.getRectangle().getMinX() + this.getRectangle().getWidth() / 2;
    }

    public double getCenterY() {
        return this.getRectangle().getMinY() + this.getRectangle().getHeight() / 2;

    }

    public double getDistance(Element e) {
        return this.getCenter().distance(e.getCenter());
    }

    public Color getStokecolor() {
        return color;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public Rectangle2D getRectangle() {
        return rectangle;
```

```
    }

    public void setRectangle(Rectangle2D rectangle) {
        this.rectangle = rectangle;
    }

    @Override
    public void debug(GraphicsContext gc) {
        //gc.setStroke(Color.RED);
        gc.setFill(Color.WHITE);

        gc.fillOval(this.getRectangle().getMinX() * Game.SCALE - 5,
                this.getRectangle().getMinY() * Game.SCALE - 5,
                10, 10);

        gc.strokeText(" X:" + (int) (this.getRectangle().getMinX())
                + " Y:" + (int) (this.getRectangle().getMinY()),
                (this.getRectangle().getMinX()) * Game.SCALE,
                (this.getRectangle().getMinY()) * Game.SCALE);

        gc.fillOval(this.getRectangle().getMaxX() * Game.SCALE - 5,
                this.getRectangle().getMinY() * Game.SCALE - 5,
                10, 10);

        gc.strokeText(" X:" + (int) (this.getRectangle().getMaxX())
                + " Y:" + (int) (this.getRectangle().getMinY()),
                (this.getRectangle().getMaxX()) * Game.SCALE + 12,
                (this.getRectangle().getMinY()) * Game.SCALE);

        gc.fillOval(this.getRectangle().getMinX() * Game.SCALE - 5,
                this.getRectangle().getMaxY() * Game.SCALE - 5,
                10, 10);

        gc.strokeText(" X:" + (int) (this.getRectangle().getMinX())
                + " Y:" + (int) (this.getRectangle().getMaxY()),
                (this.getRectangle().getMinX()) * Game.SCALE,
                (this.getRectangle().getMaxY()) * Game.SCALE);

        gc.fillOval(this.getRectangle().getMaxX() * Game.SCALE - 5,
                this.getRectangle().getMaxY() * Game.SCALE - 5,
                10, 10);

        gc.strokeText(" X:" + (int) (this.getRectangle().getMaxX())
```

```
                + " Y:" + (int) (this.getRectangle().getMaxY()),
                (this.getRectangle().getMaxX()) * Game.SCALE,
                (this.getRectangle().getMaxY()) * Game.SCALE);

        gc.fillOval(this.getCenterX() * Game.SCALE - 5,
                this.getCenterY() * Game.SCALE - 5,
                10, 10);

        gc.strokeText(" X:" + (int) (this.getCenterX())
                + " Y:" + (int) (this.getCenterY()),
                (this.getCenterX()) * Game.SCALE,
                (this.getCenterY()) * Game.SCALE);
        gc.setFill(this.color);
        gc.fillRect(this.getRectangle().getMinX() * Game.SCALE,
this.getRectangle().getMinY() * Game.SCALE, this.getRectangle().getWidth() *
Game.SCALE, this.getRectangle().getHeight() * Game.SCALE);

    }

    @Override
    public void paint(GraphicsContext gc) {
        gc.setFill(this.color);
        gc.fillRect(this.getRectangle().getMinX() * Game.SCALE,
this.getRectangle().getMinY() * Game.SCALE, this.getRectangle().getWidth() *
Game.SCALE, this.getRectangle().getHeight() * Game.SCALE);
        //gc.setStroke(getStokecolor());
        //gc.strokeRect(this.rectangle.getMinX(), this.rectangle.getMinY(),
this.rectangle.getMaxX(), this.rectangle.getMaxY());
        if (this.isDebug()) {

            this.debug(gc);
        }
    }

}
```

Para probar la clase se define como no abstracta, al procesar el nivel se crea los **“Element”** y se almacenan en un vector para pintarse:

Atributo de “Board”:

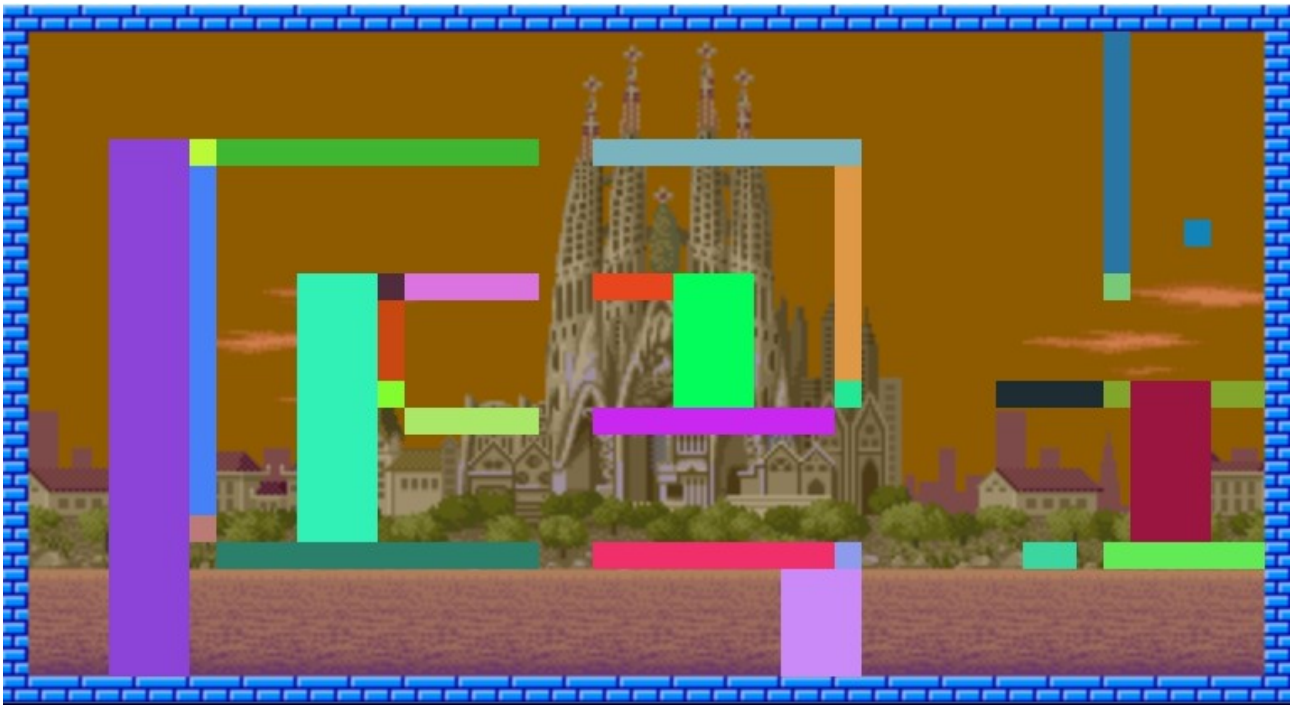
```
 private Element[] elements;
```

Al procesar el nivel:

```
private void createElementsLevel() {
     Pair<Level.ElementType, Rectangle2D>[] fi =
this.levels[this.actual_level].getFigures();
```

```
        this.elements= new Element[fi.length];
        for (int i = 0; i < fi.length; i++) {
            this.elements[i]= new Element();
            this.elements[i].setRectangle(fi[i].getValue());
           /* switch (fi[i].getKey()) {
                case FIXED:
                    break;
                case BREAKABLE:
                    break;
                case LADDER:
                    break;

            }*/
        }
}
```



3.5. Planificación.

- 1 hora: Leer la práctica.
- 1hora: Creación de las interfaces “IDebuggable” y “IDrawables”, implementación de la clase “Element”.
- 1 hora: Prueba de la clase “Element” en los diferentes niveles.
- 3 horas: Definir las clases “Ladder”y “Brick” que heredan de “Element”, modificar “Board” para crear los objetos de los tipos anteriores, y que se pinten las imágenes.

- 2 horas: Definición de la interfaz “ICollision” y la clase “ElementDynamic”. Sin implementar el método “collision”.
- 4 horas: Crear la interfaz “IMovable” y la clase “ElementMovable”, hacerla no abstracta e ir definiendo y probando los métodos.
- 3 horas: Implementar el método “collision” de la clase “ElementDynamic” y las clases que sean necesarias.
- 3 horas: Definir la clase “IGravity”, implementar la clase abstracta “ElemetWithGravity” y la “Ball” y probar.
- 1 hora: Crear una clase para almacenar las bolas en el “Board”.
- 2 horas. Definir la clase “Bros” e integrarlo en el tablero.
- 5 horas: Definir las diferentes colisiones entre los elementos.
- 3 horas: Crear las armas y municiones. Probar las colisiones y el “estallido” de las bolas.

Total 27 horas, 14 en clase.

4. Entrega.

La práctica se entrega en formato ZIP con 2 ficheros (código y presentación en formato PDF, en el campus virtual ww.aules.edu.gva.es (pendiente de matricula e inicio de curso). Se fijará la fecha en AULES.

El fichero 1 será el código generado comprimido a su vez también en zip, **importante comentar el código.**

El fichero 2 contiene la **presentación de la práctica,** que tendrá al menos los siguientes apartados. .

1. Índice.
2. Introducción.
3. URL del proyecto en GitHub (público).
4. Clases creadas.
5. Detalles más importantes de implementación de cada clase, por ejemplo.
6. Conclusiones.

5. Evaluación.

Unos días después de la entrega se realizá se realizará una presentación de 10 minutos por los alumnos en que se explicará apoyándose en la presentación presentada la realización de la práctica. Además se realizará una demostración de la funcionalidad para el resto de la clase.

Finalizada la presentación los componentes del grupo tendrán que responder a preguntas del profesor y/o resto de compañeros

- CE3d. Se ha escrito código utilizando control de excepciones.
- CE7a. Se han identificado los conceptos de herencia, superclase y subclase.
- CE7b. Se han utilizado modificadores para bloquear y forzar la herencia de clases y métodos.
- CE7c. Se ha reconocido la incidencia de los constructores en la herencia.
- CE7d. Se han creado clases heredadas que sobrescriban la implementación de métodos de la superclase.
- CE7e. Se han diseñado y aplicado jerarquías de clases.
- CE7f. Se han probado y depurado las jerarquías de clases.
- CE7g. Se han realizado programas que implementen y utilicen jerarquías de clases.
- CE7h. Se ha comentado y documentado el código.

FUNCIONALIDAD	PUNTUACIÓN
Se implementa la jerarquía desde “Element” hasta “Ball” sobreescribiendo los métodos necesarios.	2
Se define de forma completa la gestión de las armas y munición, con una jerarquía bien definida y que hace uso de las clases abstractas básicas como ElementWithResizable.	2
El jugador se mueve por el tablero, muere y dispara.	2
Se gestionan las colisiones entre los bloques, las bolas, las escaleras, las municiones y el jugador.	2
Se termina el juego cuando el jugador se queda sin vidas	0,25

Cuando se terminan las bolas se pasa de nivel	0,25
Se añade música de fondo y efectos sonoros	0.25
Se definen correctamente las interfaces	1
Comentar el código	0,25

Extras (**2 puntos**):

Se implementan a partir de ElementWithGravity las bonificaciones, asociando estas bonificaciones a “estallido” de bolas o rotura de bloques rompibles.

Toda la puntuación anterior esta supeditada a realizar la presentación y responder de forma correcta a las preguntas del profesor para comprobar la autoría del proyecto