Módulo Programación.

1º DAW.

PRÁCTICA 2: Introducción POO.
UNIDAD DE TRABAJO 4.
Profesor: Pedro Antonio Santiago Santiago.

1. Introducción.

El paradigma OO se basa en una serie de principios como son la encapsulación, la ocultación de la información o la abstracción entre otros. En esta práctica se trabaja principalmente con los principios anteriores desarrollando clases e instanciando objetos para la creación de un juego clásico PuzzleBubble.

2. Materiales.

Ordenador con JDK 11 instalado.

Netbeans 14 o superior.

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

3. Desarrollo de la práctica.

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

3.1. Descripción del juego.

Las características del juego extraídas de la Wikipedia y adaptadas a la práctica son:

El objetivo del juego es eliminar todas las burbujas que se encuentran en la parte superior de la zona rectangular en donde se encuentra el personaje. Para ello, el jugador deberá disparar burbujas desde la zona inferior con el objetivo de hacer coincidir los colores de las burbujas en **una serie de tres** o más elementos del mismo color. Tras limpiar la zona de juego, la siguiente fase comienza con un nuevo patrón de burbujas a eliminar.

Al inicio de cada ronda, la zona de juego contiene una cantidad de burbujas de colores dispuestas según un patrón predefinido. Al fondo de la pantalla, el jugador controla un artilugio denominado puntero (pointer), con el que apunta y dispara burbujas hacia arriba. El color de las burbujas a disparar se genera aleatoriamente siendo escogido de entre los colores que presentan las burbujas que todavía quedan en pantalla.

Las burbujas disparadas viajan en línea recta, pudiendo rebotar contra los laterales del rectángulo, y se detienen cuando tocan otra burbuja **o alcanzan la parte superior del rectángulo**. Cuando una burbuja choca con un grupo de dos o más burbujas de idéntico color, estas revientan y son eliminadas del campo de juego, al tiempo que se reciben puntos por la cantidad de burbujas reventadas.

Después de cierto número de disparos, el techo del rectángulo comienza a descender de forma paulatina, junto con todas las burbujas pegadas a él.

3.2. Propuesta de clases.

Se proponen las siguientes clases, con los principales atributos y métodos, pudiendo darse el caso que durante el desarrollo de la práctica sea necesaria la creación de nuevos atributos y métodos en función de las necesidades.

Además se recomienda añadir un atributo en cada clase denominado “**debug”** que cuando se encuentre activo, muestre tanto por terminal, como en el juego información de la clase, por ejemplo en la “burbuja” la dirección y la coordenada en que se está moviendo, o en el el “Shutte” o disparador, el ángulo en que se encuentra actualmente.

ESTAS PROPUESTA NO INCLUYE ANIMACIONES, EFECTOS O CUALQUIER OTRO EFECTO GRÁFICO, SOLO LA LÓGICA BÁSICA Y LOS ELEMENTOS MÍNIMOS PARA DIBUJAR LOS ELEMENTOS.

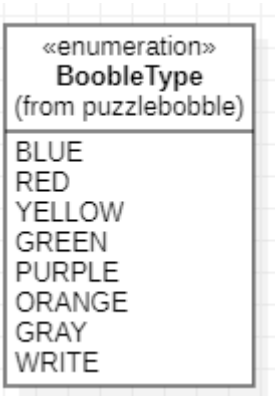
BubbleType: Enumerado con los tipos de burbujas disponibles, además se almacenan las coordenadas de la bola en la imagen.

```
public enum BubbleType {
    BLUE(1, 0),
    RED(1, 33),
    YELLOW(171,0),
    GREEN(171, 33),
    PURPLE(1, 67),
    ORANGE(171, 67),
    GRAY(1, 100),
    WRITE(171, 100);
    private final int x;
    private final int y;

    BubbleType(int x,  int y ) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

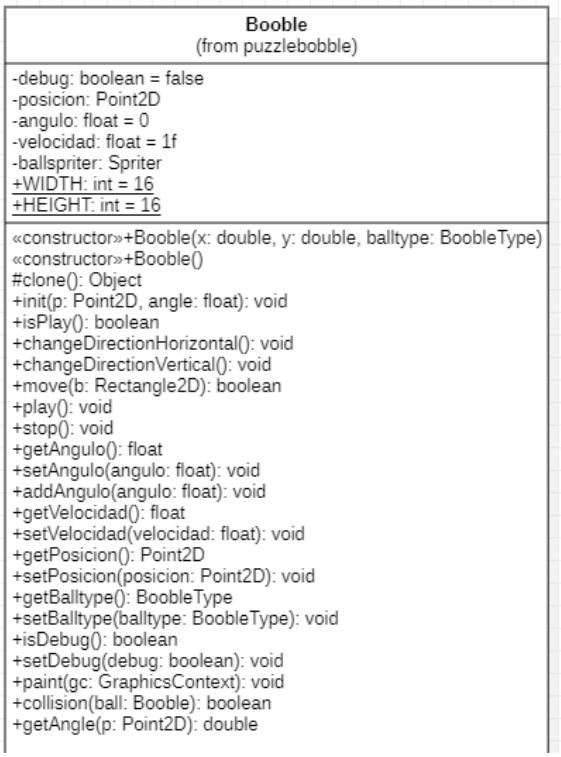


Bubble: Representa una burbuja del juego, posee un alto, un ancho, una posición,una velocidad, un estado y un tipo (determinando este el color). Los métodos principales son:

- move(Rectangle b): Mueve la burbuja, siempre que se encuentre dentro de los límites del rectángulo.
- Paint (GraphicsContext gc): Recibe un GraphicsContext que le permite pintar.
- Collision(Bubble b) boolean: Evalua si se ha colisionado con otra burbuja.
- ChangeDirectionHorizontal(): Cambia la dirección horizontal, rebota.
- ChangeDirectionVertical(): Cambia la dirección vertical, rebota.
- Posee un enumerado interno para el estado en que se encuentra la burbuja:

```
private enum State {
        PLAY,
        STOP
    }
```

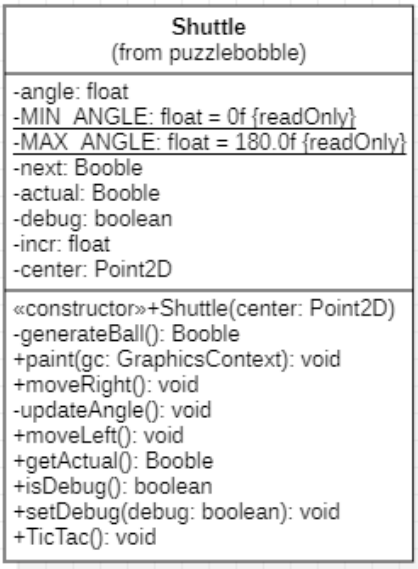
El diagrama de clases de la burbuja:



En UML + significa público, - privado, # protegido.

Shuttle: Modela la lanzadera de bolas en el juego, los atributos básicos que posee son;

- float angle: Angulo actual.
- Bubble actual: Burbuja actual preparada para disparar.
- Bubble next: Siguiente burbuja.
- Float Incr: Incremento en grados del movimiento del angulo.
- Point2D Center: Posición del disparador en el juego.
- Float MIN_ANGLE y MAX_ANGLE. Constanstes a nivel de clase con el ángulo máximo.



Los métodos más destacadas son:

- moveLeft(): Mueve el ángulo hacia la izquierda, siempre que esté entre los márgenes.
- MoveRigth(): Idem que el anterior.
- Shoot() Bubble: Devuelve la burbuja actual, pasa la siguiente a actual y genera una nueva que sera la de reserva, la burbuja que devuelve tiene la posición correcta y el ángulo actual del Shuttle.

BallGrid: Una de las clases más importantes, es una matriz de burbujas que se encuentran pegadas a la parte superior del juego. Sus principales funciones son:

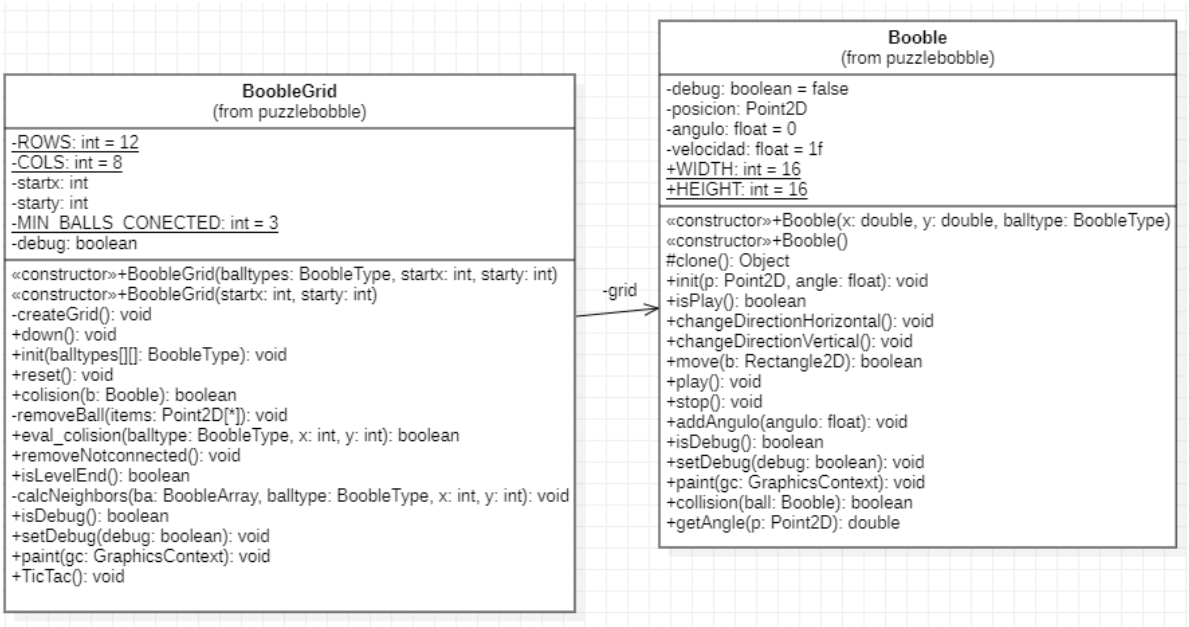
- Detectar colisiones con la burbuja que se está moviendo.

- Añadir en caso de colisión la burbuja que se mueve a la matriz en la posición correcta.
- Detectar cuando se produzca una colisión si existen más de 3 burbujas contiguas y eliminarlas.
- Detectar cuando se eliminen burbujas, si se han quedado otras solas para eliminarlas.
- Evaluar si se ha terminado el juego.
- Evaluar si se pasa de nivel.

Los atributos de esta clase son:

- int startx: Posición de inicio de la pantalla en la coordenada x (va bajando a medida que evoluciona el juego.
- Int Starty: Idem que el anterior en la posición y, no cambia en el juego.
- Bubble grid[][]: Matriz de burbujas, si la celda no tiene burbuja el valor es null, si la tiene es por el valor inicial, o que se ha añadido a lo largo del juego.
- ROWS,COLS y MIN_BALLS_CONECTED, son atributos estáticos con las dimensiones de la matriz y número mínimo de burbujas conectadas.

El diagrama de clases:



Destacando los métodos:

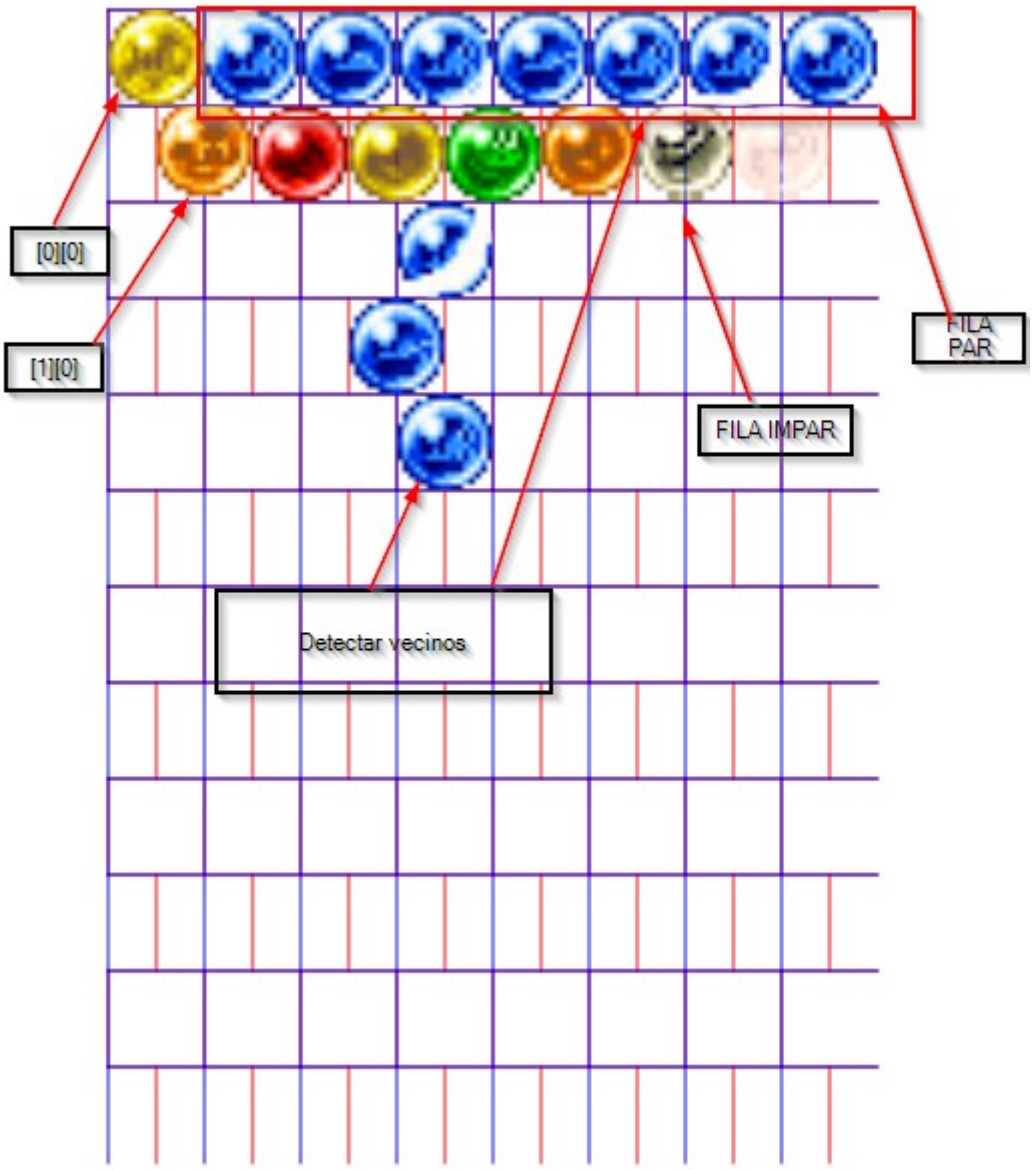
- void init(BubbleType balltypes[][]): Inicializa la matriz a partir de una matriz de tipos de burbuja, es decir los niveles.
- Boolean eval_colision(BubbleType balltype, int x, int y): A partir de la posición x e y y el tipo de Burbuja, evaluá si se han de explotar esa burbuja y las aledañas.
- void removeNotconnected(): Elimina las burbujas que no se encuentran conectadas con la parte superior.
- boolean isLevelEnd(): Cierto en caso de que no queden burbujas en la matriz.
- Boolean isEnd(): Cierto si la burbuja inferior ha llegado a la parte inferior.
- void calcNeighbors(BubbleArray ba, BubbleType balltype, int x, int y). Función recursiva que calcula los vecinos del mismo tipo de una burbuja. Se le pasa un objeto de tipo BubbleArray, que es una vector dinámico primitivo, se explicará en clase.
- boolean collision(Bubble b). Cierto si alguna de las burbujas colisiona con la que se pasa por parámetro.
- void down(): Baja la matriz una fila hacia abajo, es decir, modifica stary.

La matriz es irregular, las filas pares (0,2….) poseen 9 columnas, y las filas impares 8.

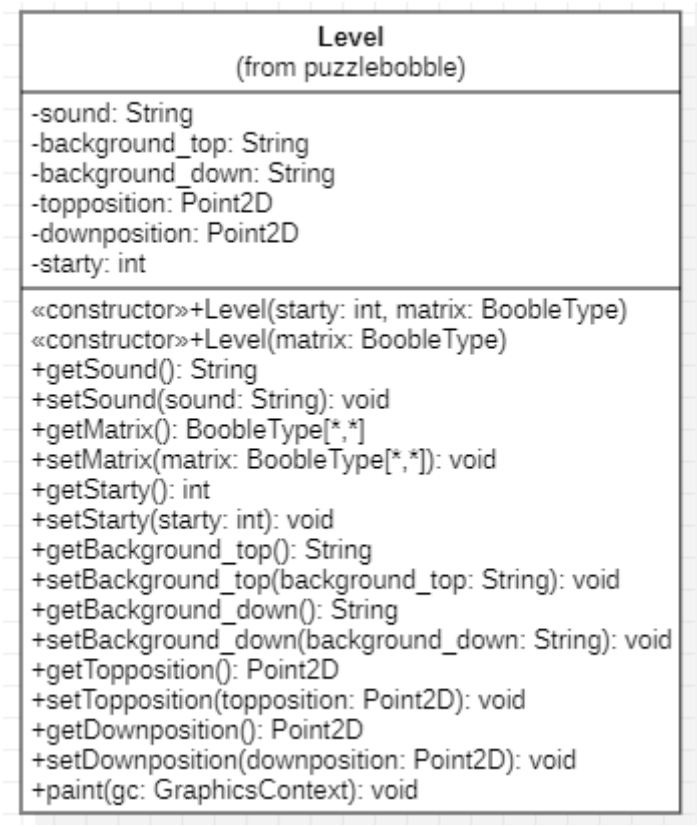
Cuando se dibuja la matriz, las filas pares se pintan en la posición x: startx y las impares desplazadas a la derecha.

En la imagen siguiente se pueden ver con líneas azules, las posiciones naturales de la matriz, y en rojo, el desplazamiento (se produce en las impares), además de las situaciones en la que se ha de detectar los vecinos.

El desplazamiento se tendrá que tener en cuenta para realizar los diferentes cálculos, como el cálculo de vecinos.



Level: Clase que representa un nivel del juego, simplemente posee una matriz de BubbleType, que se le pasará al BubbleGrid para crear el juego, la posición inicial x e y, la música que sonará en el juego y las imágenes de fondo a mostrar.



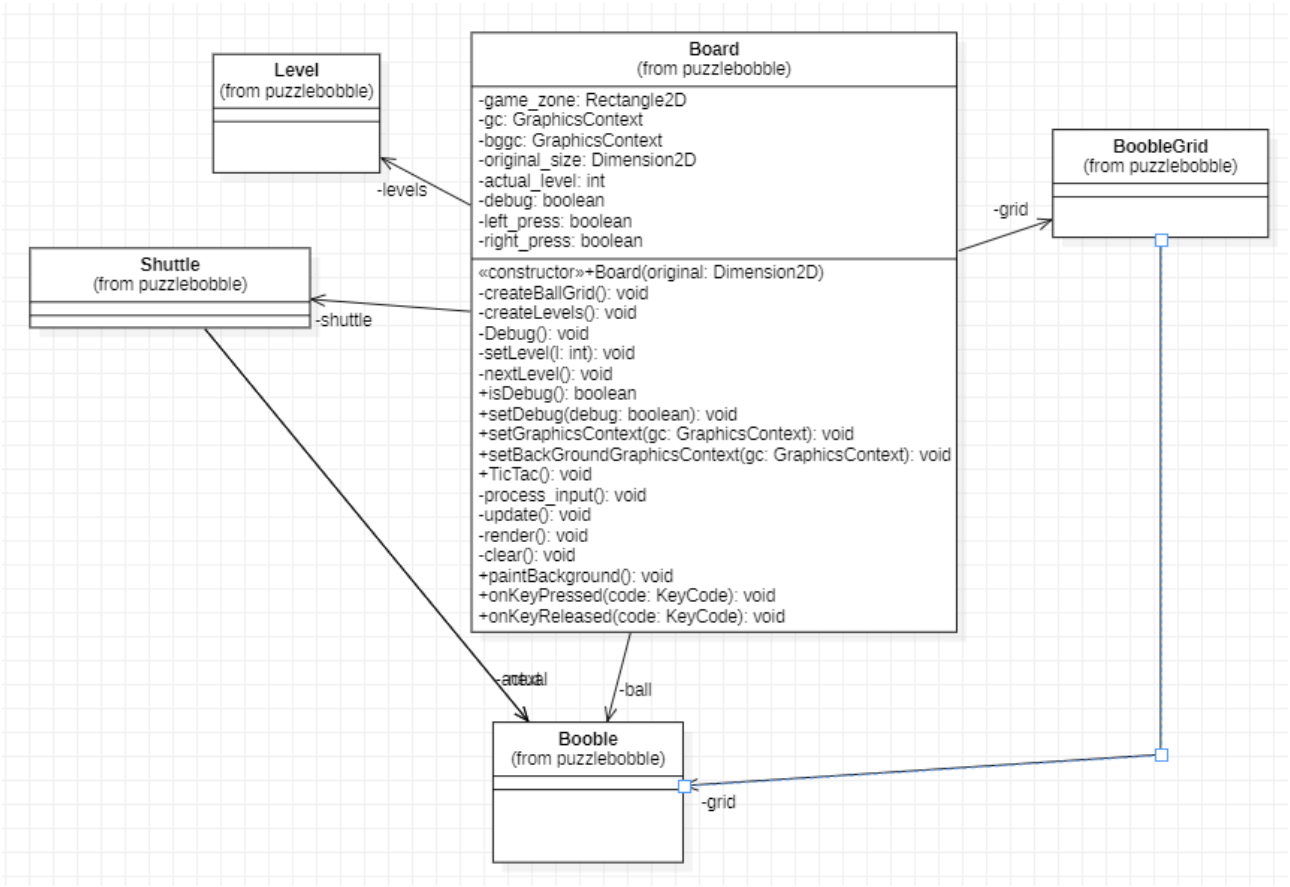
Board: Tablero del juego, contiene los niveles, el grid de burbujas y el disparador, se encarga de la lógica del juego.

Los atributos principales son:

- Rectangle2D game_zone. Área en pixels del juego, del total de la pantall
- GraphicsContext gc. Para pintar en el canvas principal
- GraphicsContext bggc. Para pintar en el canvas de fondo.
- Dimension2D original_size. Tamaño original de las imágenes, permite realizar un escalado
- Bubble ball. Burbuja que se mueve por el juego
- Level levels[]. Vector de niveles
- int actual_level. Nivel actual
- Shuttle shuttle. Disparador
- BubbleGrid grid.

- boolean left_press, right_press. Cierto o falso si las teclas se encuentran pulsadas, para mover el “Shutter”.

En la siguiente imagen se puede ver la clase Board y la relación con el resto de las clases



Los métodos más destacados:

- void createLevels(). Crea los niveles.
- Void setLevel(int l). Pone el nivel de la posición l del vector como actual.
- Void nexLevel(): Pasa al siguiente nivel.
- synchronized void TicTac(). Método que es ejecutado por el reloj, contiene el análisis de la entrada, la actualización del juego y la actualización de la pantalla.
- void paintBackground(): Pinta la imagen de fondo a partir del nivel actual.
- void onKeyPressed(KeyCode code). Cada vez que se pulsa una tecla se pasa el código a este método.
- void onKeyReleased(KeyCode code). Igual que el anterior, pero cuando se suelta.

Otras clases que se facilitan para el juego son:

- **Clock.** reloj que tiene una lista de escuchadores (se tratará en temas posteriores), cada vez que pasa un cierto tiempo, se ejecuta el método TicTac de la lista.
- **Game.** El juego propiamente dicho, tiene un atributo **estático público con la escala** (para pintar en diferentes tamaños), una imagen también estática y pública para que se pueda usar en el resto de las clases, un reloj, los "canvas" de primer y segundo plano y un "Board" tablero.
- **App**. Clase con las dimensiones del juego, los "canvas" (áreas de dibujo) y se encarga de pasar las pulsaciones del teclado al tablero.
- **Resources**: Almacena los recursos multimedia, carga en memoria una única vez todos, estos recursos se encuentran en la carpeta: src\main\resources. Cada recurso (fichero) se asocia con un nombre, devolviendo objetos de tipo Image o tipo MediaPlayer (sonidos).Por defecto se cargan:
  - Imágenes:

    {"fondos", "escenarios.png"},

    {"spriters", "todos.png"},

    {"wheel", "ruleta.png"},

    {"arrow", "flechas.png"},

    {"dragons", "dragons.png"},

    {"balls", "balls.png"}
  - Sonidos:

    {"pared", "BallBounce.wav"},

    {"disparo", "BubbleShot.wav"},

    {"fondo", "SinglePlayerMusic.wav"}

  Para obtener una imagen:

```
Resources.getInstance().getImage(String name);
```

Para obtener un MediaPlayer:

```
Resources.getInstace().getSound(String name);
```

3.3. Proyecto inicial.

Se facilita un proyecto inicial con las clases:

- Game
- Clock
- Board. Básico.
- Resources.
- App.

Además se modifica el fichero pom.xml para poder depurar de forma remota, para depurar en la pestaña Navigator en el proyecto, buscar el "Goal" javafx run y añadir @debug en la ejecución con modificadores.

3.4. ¿Cómo empezar?.

3.4.1. La carpeta "resources" y cómo usarla.

Ya se ha indicado que se tiene una carpeta llamada resources en src/main/resources y que la clase "Resources" que implementa el patrón "Singleton". Ver vídeo https://www.youtube.com/watch?v=GGq6s7xhHzY.

Si se quiere añadir nuevas imágenes o sonidos, colocarlos en esa carpeta y añadirlos en el vector de imágenes/sonidos de la clase Resource. Para obtenerlo simplemente llamar a getInstance().getImage(String name) o getInstance().getSound().

Se proporcionan las imágenes:

- ball.png. Imagen que contiene diferentes burbujas en diferentes colores
- dragons.png. Contiene los dos dragones, con imágenes del movimiento.
- Escenario.png. Los diferentes escenarios.
- Flechas.png. Animación del "shutte".
- Ruleta.png. Animación de la ruleta.
- Todos.png. Todos los elementos junto con sus animaciones.

Se opta por dividir en varios para facilitar el posicionamiento, aunque se puede usar solo la imagen todos.png. Si se considera oportuno se pueden añadir más imágenes.

En cuanto al sonido, se tiene:

- BallBounce.wav.
- BubleShot.wav
- SinglePlayerMusic.wav.

3.4.2.Dibujar.

Se puede dibujar en 2 lugares, el primero el fondo, que solo se pinta cada vez que se cargé un nuevo nivel ya que no tiene sentido estar pintando una y otra vez una imagen que no cambia y el primer plano que se dibuja en cada ciclo.

Cada uno de estos canvas dispone de un "GraphicContext" que permite dibujar diferentes formas (líneas, ovalos…) e imágenes.

Para dibujar se obtiene la imagen de "Resource", por ejemplo las burbujas:

```
Resources r = Resources.getInstance();
Image m=r.getImage("balls");
```

Y en un objeto de tipo GraphicsContext se indica.

- Posición x de inicio en la imagen origen.
- Posición y de inicio en la imagen origen.
- Ancho seleccionado en la imagen origen.
- Alto seleccionado en la imagen origen.
- Posición x en el lienzo.

- Posición y en el lienzo.
- Ancho en el lienzo.
- Alto en el lienzo.

Por ejemplo se desea “pintar” el fondo en el objeto “Board”, los fondos se encuentra en recurso “escenarios”, y el primero empieza en la posición (pixels):

x: 16

y:17

El tablero tienen un alto y ancho definido en App, que se pasa a “Board” en el constructor en un objeto de tipo Dimension2d que se almacena en el atributo original_size de “Board”.

ancho:320

alto:224

Además se tiene una escala en en Game.SCALE de 2 (puede ser de 1,1.5.3….). Con lo que se ha de dibujar el fragmento de “escenarios.png” llamados “fondos” que empieza en 16,17, con ancho 320, 224, en el lienzo en la posición 0,0, con un ancho 320*ESCALA y un alto 224*ESCALA.

En código:

```
  public void paintBackground() {
        Resources r = Resources.getInstance();
        Image fondos=r.getImage("fondos");
        //se dibujar el fondo
        this.bggc.drawImage(fondos,
                //inicio de la posicion
                16, //incio en x de la imagen original
                17, //inicio en y de la imagen origina
                this.original_size.getWidth(), //ancho de la imagen original
                this.original_size.getHeight(), //alto de la imagen origina
                //dibujar en el liendozo
                0, //pociión x en el lienzo
                0, //posición y en el lienzo
                this.original_size.getWidth() * Game.SCALE, //ancho en el
lienzo
                this.original_size.getHeight() * Game.SCALE); //alto en el
lienzo
```

```
        //se dibuja la línea del fondo
        if (this.debug) {
            this.Debug();
        }
    }
```

3.4.3.Sonidos.

Aunque la práctica no lo pide, existe la posibilidad de tener ejecutar diferentes sonidos en función de las necesidades

Se procede de forma similar a las imágenes, se obtiene el MediaPlayer.

```
MediaPlayer sonido = Resources.getInstance().getSound("fondo");
//iniciar
sonido.play();
//parar
sonido.stop();
```

3.4.4.Empezando el proyecto. La burbuja.

Al igual que el resto de los proyectos, y en el desarrollo en general, se ha de comenzar por las tareas más sencillas, y de forma incremental ir añadiendo clases y algoritmos más complejos. **Hasta que no se ha solucionado un problema no se pasa al siguiente, ya que hacer esto hace el proyecto inmanejable.**

Se define la clase Bubble que contiene:

- Point2D Posición. En este caso se utiliza una clase de JavaFx Point2D.
- Float Angulo. Ángulo de la pelota.
- Float Velocidad. La velocidad tiene que ser menor o igual de 1, es decir, en el mejor de los casos, en un ciclo puede avanzar un pixel en el eje x o un pixel en el eje y.
- Int ANCHO. Estático para todas las burbujas.
- Int ALTO. Estático para todas las burbujas.
- Un enumerado con los posibles estados de la burbuja: PLAY y STOP.
- Un enumerado con los tipos de burbujas: ROJO,AZUL.

En cuanto que puede hacer una burbuja, se ha indicado ya en puntos anteriores:

- move(Rectangle b): Mueve la burbuja, siempre que se encuentre dentro de los límites del rectángulo.
- Paint (GraphicsContext gc): Recibe un GraphicsContext que le permite pintar.
- Collision(Bubble b) boolean: Evalua si se ha colisionado con otra burbuja.
- ChangeDirectionHorizontal(): Cambia la dirección horizontal, rebota.
- ChangeDirectionVertical(): Cambia la dirección vertical, rebota.

Se crea la clase Bubble, con los atributos y métodos anteriores, destacar el método mover al que se le pasa un rectángulo, detecta colisiones con las paredes y cambia la dirección, y el método paint,además del método “collision” que no se implementa inicialmente.

```
public class Bubble {
    private enum State {
        PLAY,
        STOP
    }
    private boolean debug = false;
    private State estado;
    private Point2D posicion;
    private float angulo = 0;
    private float velocidad = 1f;
    private BubbleType balltype;
    public static int WIDTH = 16, HEIGHT = 16;

    public Bubble(double x, double y, BubbleType balltype) {
        this.estado = State.STOP;
        this.posicion = new Point2D(x, y);
        this.balltype = balltype;
    }

    public Bubble() {
        this.estado = State.STOP;

    }

    public void init(Point2D p, float angle) {
        int vertical_center = (int) (p.getX());
        int horizontal_center = (int) (p.getY());
```

```
        this.angulo = angle;
        this.posicion = new Point2D(vertical_center, horizontal_center);
        BubbleType[] balltypes = BubbleType.values();
        this.setBalltype(balltypes[(int) (Math.random() * balltypes.length)]);
        this.stop();
    }

    public boolean isPlay() {
        return this.estado == State.PLAY;
    }

    public void changeDirectionHorizontal() {
        this.setAngulo(180.0f - this.getAngulo());
    }

    public void changeDirectionVertical() {
        this.setAngulo(360.0f - this.getAngulo());
    }

    public boolean move(Rectangle2D b) {
        boolean pared = false;
        if (this.estado == State.PLAY) {
            float x = (float) ((float) getVelocidad() *
Math.cos(Math.toRadians(getAngulo())));
            float y = (float) ((float) getVelocidad() *
Math.sin(Math.toRadians(getAngulo())));
            this.posicion = this.getPosicion().add(x, y);
            //izquierda
            if (this.getPosicion().getX() - (Bubble.WIDTH / 2) < b.getMinX()) {
                pared = true;
                this.changeDirectionHorizontal();
            } else {

                //derecha
                if (this.getPosicion().getX() + (Bubble.WIDTH / 2) >
b.getMinX() + b.getWidth()) {
                    pared = true;
                    this.changeDirectionHorizontal();

                } else {
                    //parte inferior
```

```
                    if ((this.getPosicion().getY() + (Bubble.HEIGHT / 2)) -
(b.getMinY() + b.getHeight()) >= 0) {
                        pared = true;
                        this.changeDirectionVertical();
                    } //parte superior, no debe rebotar
                    else if (this.getPosicion().getY() - (Bubble.HEIGHT / 2) <=
b.getMinY()) {
                        pared = true;
                        this.changeDirectionVertical();

                    }
                }
            }

        }
        return pared;

    }

    public void play() {
        this.estado = State.PLAY;
    }

    public void stop() {
        this.estado = State.STOP;
    }

    /**
     * @return the angulo
     */
    public float getAngulo() {
        return angulo;
    }

    /**
     * @param angulo the angulo to set
     */
    public void setAngulo(float angulo) {
        this.angulo = angulo;
        if (this.angulo < 0) {
            this.angulo += 360;
```

```
        }
        if (this.angulo >= 360) {
            this.angulo = this.angulo % 360;
        }
    }

    public void addAngulo(float angulo) {
        this.angulo += angulo;
        if (this.angulo >= 360) {
            this.angulo = this.angulo % 360;
        }
        if (this.angulo < 0) {
            this.angulo += 360;
        }
    }

    /**
     * @return the velocidad
     */
    public float getVelocidad() {
        return velocidad;
    }

    /**
     * @param velocidad the velocidad to set
     */
    public void setVelocidad(float velocidad) {
        this.velocidad = velocidad;
    }

    /**
     * @return the posicion
     */
    public Point2D getPosicion() {
        return posicion;
    }

    /**
     * @param posicion the posicion to set
     */
```

```
    public void setPosicion(Point2D posicion) {
        this.posicion = posicion;
    }

    /**
     * @return the balltype
     */
    public BubbleType getBalltype() {
        return balltype;
    }

    /**
     * @param balltype the balltype to set
     */
    public void setBalltype(BubbleType balltype) {
        this.balltype = balltype;
    }

    /**
     * @return the debug
     */
    public boolean isDebug() {
        return debug;
    }

    /**
     * @param debug the debug to set
     */
    public void setDebug(boolean debug) {
        this.debug = debug;
    }

    public void paint(GraphicsContext gc) {

        Resources r = Resources.getInstance();

        gc.drawImage(r.getImage("balls"),
                //inicio de la posicion
                this.getBalltype().getX(),
                this.getBalltype().getY(),
```

```
                Bubble.WIDTH,
                Bubble.HEIGHT,
                //dibujar en el lienzo
                (this.posicion.getX() - Bubble.WIDTH / 2) * Game.SCALE,
                (this.posicion.getY() - Bubble.HEIGHT / 2) * Game.SCALE,
                Bubble.WIDTH * Game.SCALE,
                Bubble.HEIGHT * Game.SCALE);
        //si se esta depurando
        if (this.debug) {
            gc.setStroke(Color.RED);
            gc.fillOval(this.getPosicion().getX() * Game.SCALE - 5,
(this.getPosicion().getY()) * Game.SCALE - 5, 10, 10);
            gc.setStroke(Color.GREEN);
            gc.strokeText(this.angulo + "° x:" + this.getPosicion().getX() + "
y:" + this.getPosicion().getY(), (this.getPosicion().getX() - WIDTH / 2) *
Game.SCALE, (this.getPosicion().getY() - HEIGHT / 2) * Game.SCALE);
        }
    }

    public boolean collision(Bubble ball) {
            return true;
    }

    public String toString() {
        return "x:" + this.posicion.getX() + " y:" + this.posicion.getY() + "
angulo:" + this.angulo + " w:" + Bubble.WIDTH + " h:" + Bubble.HEIGHT;
    }
}
```

Ahora queda probar el código, para ello se define en “Board” el área de juego, que será un atributo de tipo Rectangle2D (incluido en JavaFx) llamado “gamezone” :

```
  public Board(Dimension2D original) {
        this.gc = null;
        this.game_zone = new Rectangle2D(95, 23, 128, 200);
        this.original_size = original;
        this.right_press = false;
        this.left_press = false;
        this.debug = false;

    }
```

Se define un atributo en el tablero (después se tendrá que quitar) que será una burbuja para probar:

```
private Bubble ball;
```

Cuando se pulse “espacio” se creará una nueva burbuja:

```
public void onKeyReleased(KeyCode code) {
        switch (code) {
            case LEFT:
                this.left_press = false;
                break;
            case RIGHT:
                this.right_press = false;
                break;
            case ENTER:
                break;
            case SPACE:
                //se crea
                this.ball=new Bubble();
                //se coloca el tipo de forma aleatorioa
                this.ball.setBalltype(BubbleType.values()[     (int)
(Math.random()*BubbleType.values().length)]);
                //se pone la posición (centro) y ángulo aleatorio
                this.ball.init(new Point2D(
                (this.game_zone.getMaxX() - this.game_zone.getWidth() / 2),
                        (this.game_zone.getMaxY() - 18)

                ), (float) (Math.random()*360));
                 this.ball.play();
                break;
            case P:


                break;
            case E:
                break;
             case D:
                this.setDebug(!this.debug);
                this.paintBackground();
        }
    }
```

Y en cada ciclo del juego se moverá y pintará:

```
public synchronized void TicTac() {
```

```
        this.clear();
        this.render();
        this.process_input();
        this.update();
    }
    private void update() {
        //actualizar el juego
        if (this.ball != null && this.ball.getBalltype() != null) {
            this.ball.move(this.game_zone);
        }
    }
    private void render() {
        if (this.ball != null && this.ball.getBalltype() != null) {
            this.ball.paint(gc);
        }
    }
    private void process_input() {
        if (this.left_press) {

        } else if (this.right_press) {

        } else {

        }
    }
```

3.5. Recomendaciones.

1. Crear clases de una en una integrándolas a medida que el proyecto avanza, un posible orden:
    1. Bubble.
    2. Shotter.
    3. GridBubble.
    4. Level.

Por ejemplo cuando se tiene la burbuja, se continua con el disparador, en primer lugar conseguir que su ángulo cambie al pulsar las teclas y en segundo que al pulsar una tecla se dispare (en caso de poder) y se genere una nueva burbuja en la recámara.

2. Dar pasos lo más pequeños posibles.
3. Diseñar en papel los métodos más complejos como detectar colisiones entre burbujas, calcular vecinos al colisionar con el grid o calcular si en el “grid” se quedan burbujas sueltas.
4. Ir añadiendo al tablero y al juego las nuevas funcionalidades a medida que se crean, como pasar de nivel o detectar fin de juego.

3.6. Temas transversales.

Durante el desarrollo de las clases, en oficinas y en casa muchos de los equipos se colocan en standby por comodidad, ya que se facilita su puesta en marcha. Esta comodidad tiene su coste, y no es menor, tanto económico como especialmente ambiental.

A partir de la calculadora proporcionada por la asociación de consumidores, https://www.ocu.org/vivienda-y-energia/nc/calculadora/consumo-en-stand-by:

Aparatos de ofimática

		Potencia en Stand-by	Consumo anual	Gasto anual	CO2 producido
☑	Ordenador ⓘ	5	44	7,45	28,5
☑	Portátil ⓘ	4	35	5,96	22,8
☑	Monitor CRT ⓘ	3	26	4,47	17,1
☑	Monitor LCD ⓘ	1	9	1,49	5,7
☑	Router	8	70	11,91	45,6
☑	Impresora	8	70	11,91	45,6
☑	Altavoces PC	3	26	4,47	17,1

Aparatos de cocina

1. Calcular el coste para el centro de las aulas del ciclo formativo y lo que es más importante el CO2 producido.
2. Calcular el consumo y CO2 producido por los aparatos que se encuentran en "standby" en casa.
3. Comentar con los compañeros alguna posible solución para este problema.

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. Cada una de las partes en un proyecto Maven.**

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.
    - Inicialización del juego.
    - Algoritmo de colisión con el "grid".
    - Algoritmo de detección de número de vecinos con el mismo color.
    - Algoritmo de detección de burbujas "solas" al romper otras.
    - Declaración de atributos y métodos estáticos.
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

- CE2a. Se han identificado los fundamentos de la programación orientada a objetos.
- CE2b. Se han escrito programas simples.
- CE2c. Se han instanciado objetos a partir de clases predefinidas 1
- CE2d. Se han utilizado métodos y propiedades de los objetos.
- CE2e. Se han escrito llamadas a métodos estáticos.
- CE2f. Se han utilizado parámetros en la llamada a métodos.
- CE2g. Se han incorporado y utilizado librerías de objetos.
- CE2h. Se han utilizado constructores.
- CE2i. Se ha utilizado el entorno integrado de desarrollo en la creación y compilación de programas simples.
- CE4a. Se ha reconocido la sintaxis, estructura y componentes típicos de una clase.
- CE4b. Se han definido clases.
- CE4c. Se han definido propiedades y métodos.
- CE4d. Se han creado constructores.
- CE4e.Se han desarrollado programas que instancien y utilicen objetos delas clases creadas anteriormente.
- CE4f. Se han utilizado mecanismos para controlar la visibilidad de las clases y de sus miembros.
- CE4h. Se han creado y utilizado métodos estáticos.

- CE4j. Se han creado y utilizado conjuntos y librerías de clases.

FUNCIONALIDAD	PUNTUACIÓN
La burbuja actual se mueve correctamente, chocando y cambiando de dirección	0,25
El disparador cambia el ángulo entre unos valores máximos y mínimos definidos	0,5
El disparador dispara correctamente y general la siguiente	0,75
Se crea el grid de burbujas	1,25
La burbuja colisiona con el grid y se coloca en la posición correcta.	1,75
Al colisionar elimina cuando se tenga n o más burbujas del mismo color comentadas	2
Elimina las burbujas sueltas	1
Define diferentes niveles	1,25
Comenta el código	0,25
Pasa de nivel cuando se queda sin burbujas	0,5
Detecta la finalización del juego.	0,5

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