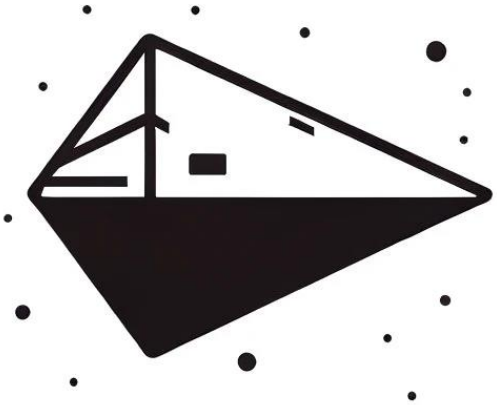
TGPCT Manual

V 1.0



Alejandro Contell

acontellfernandez@cifpfbmoll.eu

13 de Enero de 2024

CIFP Francesc B Moll

Índice

Estructura general

El programa se estructura en una serie de directorios y archivos, dentro del UML se pueden diferenciar por el texto de las flechas, con excepción a la palabra *implements* y *extends*. Cada una que contenga un texto como *TGR* indica el paso a un directorio diferente. Además, en casos como *Rules* u *Objects* se explicita su estructuración en directorios.

El código presentado se corresponde con la sección azul del UML, que representa el modelo y su controlador, además de la parte visual de la aplicación. Cada flecha roja representa una clase que implementa *Runnable* y constituirá un hilo diferente del principal (*Main*).

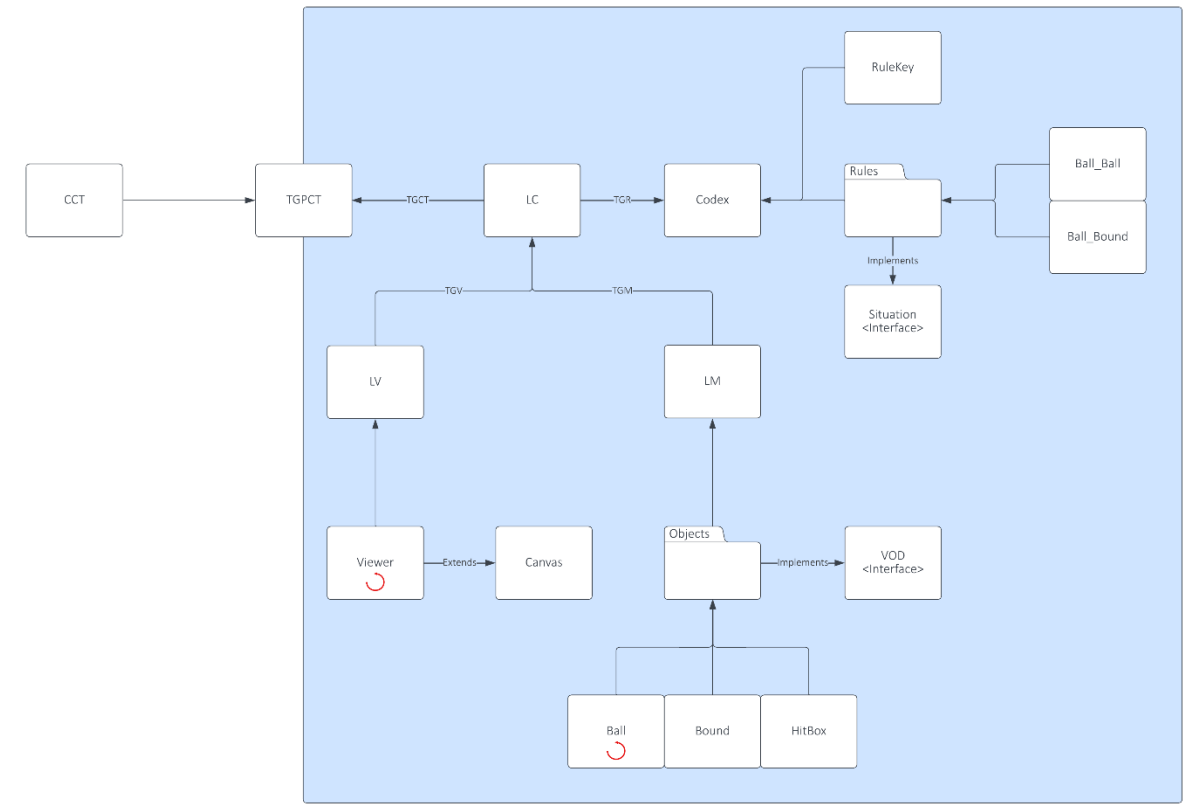


Ilustración 1. UML.

Inicio de la aplicación

Toda la aplicación se inicia desde el *TCPCT*, donde se especifica el ancho y alto deseado, y se pasan como atributos la *LC* o *Local Controller*, que una vez recibidos instancia el *Local Viewer* (*LV*) y el *Local Model* (*LM*) y, sumado a los atributos anteriores, les pasa su instancia actual para permitir la comunicación entre módulos.

A partir de este punto, cada parte se inicializa de forma independiente. Empezando por el *LM*, este recibe como atributos en el constructor, el ancho, alto, y la instancia del *LC*. Una vez recibidos, crea un *ArrayList()* para guardar los objetos que se irán generando y crea los bordes (*Bounds*) que limitarán la sección donde dichos objetos podrán actuar. Cabe recalcar que estos bordes ocupan 2px en todas direcciones, de forma que, si la ventana tiene un tamaño de 600x600, la sección jugable tendrá unas dimensiones de 596x596.

En el lado contrario se inicia la sección visual de la aplicación a través del *LV*, este recibe las dimensiones de la ventana e instancia el *Viewer*, encargado de actualizar la parte visual y de ejecutar la creación de objetos al pulsar el ratón (*mouseListener()*). Además, el *LV* inicia el hilo del *Viewer*, permitiendo una actualización independiente del hilo principal (*main*).

Por último, tenemos el *Codex*, encargado del reglamento y resolución de conflictos en el *LM*. Este ejecutará el *setCodex()* creando todas las entradas del reglamento junto a cómo resolver cada situación como un impacto con los bordes o con otro objeto.

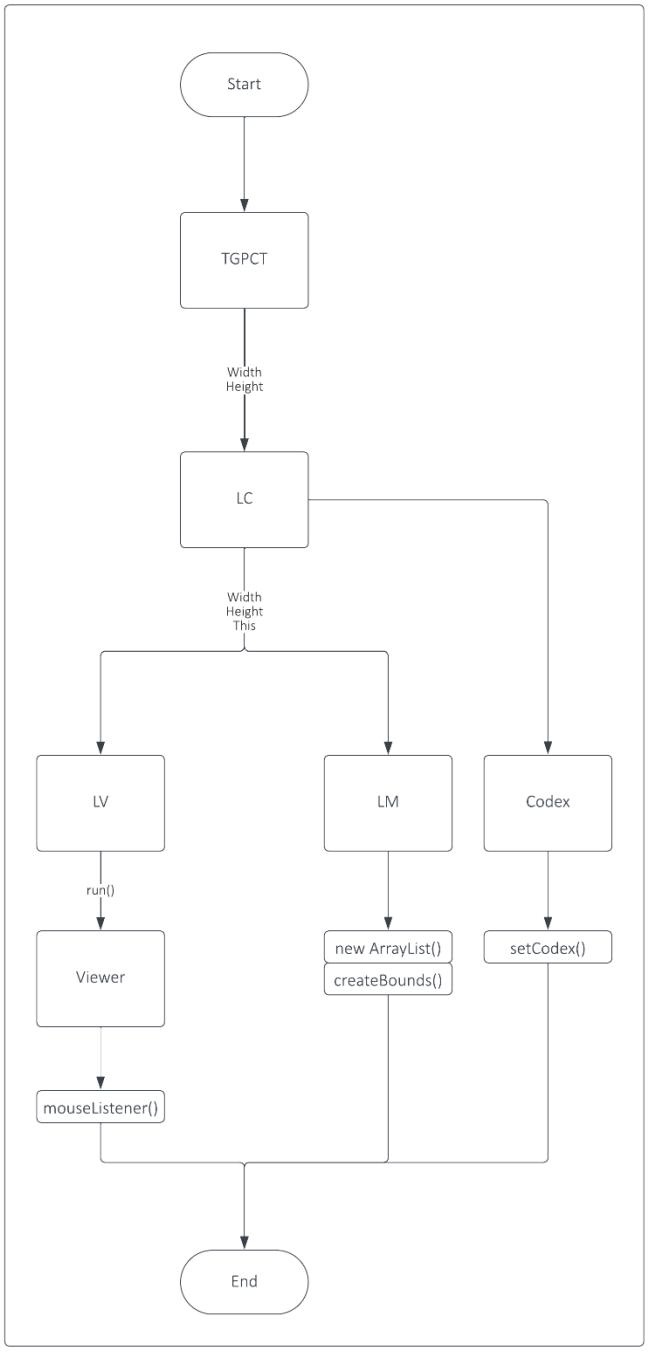


Ilustración 2. Inicio de la aplicación.

Proceso de vida de un objeto

Dentro del programa, cada objeto dinámico (*Visual Object Dynamic – VOD*) como es el caso de *Ball* opera siguiendo una serie de instrucciones.

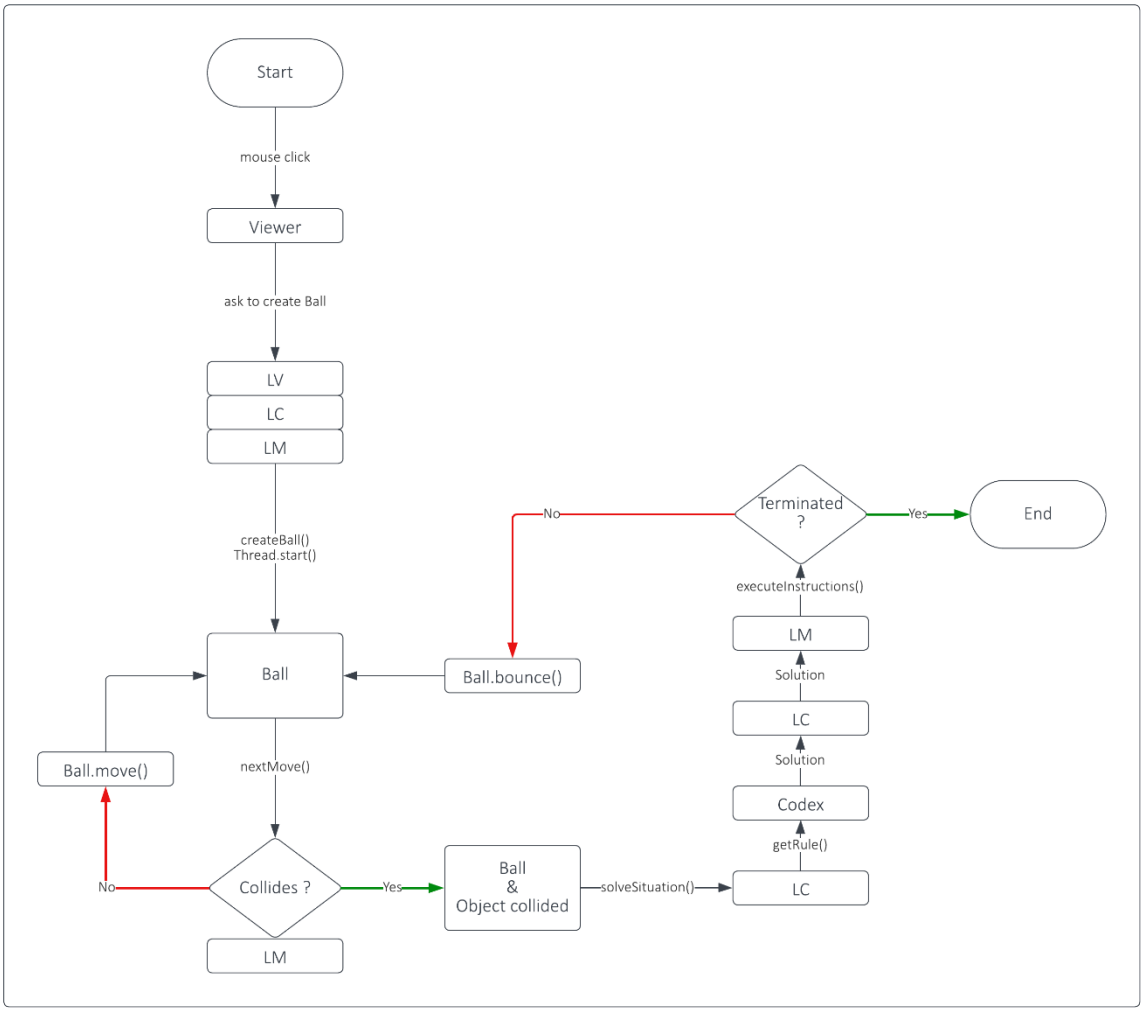


Ilustración 3. Funcionamiento de VODs.

Centrándonos en el caso actual, en el momento en el que se pulsa el botón ratón izquierdo dentro del área que ocupa el *Viewer* en la aplicación, este lo detecta y solicita al *Local Viewer* (*LV*) que cree una *Ball* en el punto (x, y) donde se ha pulsado, el *LV* a su vez, recibe estas coordenadas y se las pasa al *Local Controller* (*LC*), el cual hace lo propio y pide al *Local Model* (*LM*) que ejecute el método *createBall()* en las coordenadas extraídas en el *Viewer*.

En este momento se ha creado un objeto *Ball* en la posición en la que se realizó el clic del ratón. Los objetos *Ball* se rigen en base a unas velocidades y dinámica propia (Ver apartado *Ball* para más información), pero, en resumen, en el momento de crearse se les asigna una velocidad y dirección aleatoria y empiezan su ciclo de vida.

Cuando una *Ball* quiere moverse, procede de la siguiente forma:

1. *Ball* ejecuta el cálculo de las nuevas coordenadas y se las pasa al *LM*, en caso de que no colisione con ningún otro objeto el *LM* permite a *Ball* ejecutar *move()* y moverse a la posición calculada, pero en caso de que colisione, *LM* identificará el objeto con el que va a colisionar y pide al *LC* como debe proceder.
2. *LC*, que recibe por parámetro los objetos que intervienen en la colisión, se los pasa al *Códex* y le pregunta cómo debe proceder. Este revisará en las normas como proceder con la pareja actual de objetos y le dará al *LC* un *String* (*Solution* en la *ilustración 3*) y este otro se lo devolverá al *LM*.
3. Una vez el *LM* ha recibido la forma de proceder, ejecuta el método *executeInstructions()* y le dice a cada objeto involucrado como debe proceder. En este punto, *Ball* ha recibido una instrucción que puede ser *termiante()*, en cuyo caso se autodestruye, o *bounce()*, en cuyo caso rebota en función de un código (*String*) que requiere el método para decidir de qué forma rebotar.
4. Después de ejecutar *bounce()* se vuelve a empezar el bucle.

Funcionamiento en detalle

Por motivos de legibilidad y redundancia se ha omitido la mención a los *setters* y *getters* debido a su uso constante y simpleza dentro de la aplicación.

Visual Object Dynamics – VOD

Esta interfaz se usa como modelo para el resto de los objetos que la implementen, denominados dinámicos, pues esta está relacionada con movimientos, colisiones y apartado gráfico. Establece los métodos básicos necesarios para el correcto funcionamiento de los *VOD* de forma que facilite el desarrollo de estos. Poco más a mencionar, es una plantilla que solo fuerza a tener unos métodos, cuyas acciones se establecen en los propios objetos.

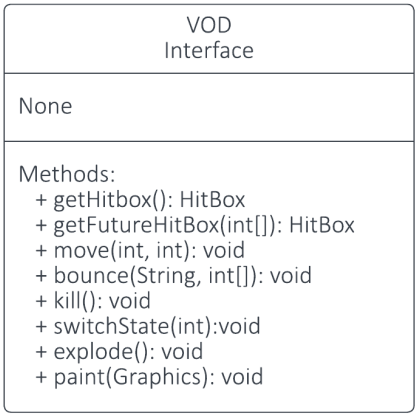


Ilustración 4. VOD en detalle.

Ball

Atributos

Se constituye por tres tipos de atributos, físicos, estado y modelo. Cada tipo está dedicado a una sección específica del funcionamiento del objeto.

- Los atributos físicos constituyen lo necesario para el cálculo de nuevas coordenadas y la resolución de problemas, además, son usados para la visualización de este en el *Viewer*.
- El atributo de estado se usa para describir la situación actual del objeto como podría ser *alive* o *termianted*.
- El atributo de modelo se usa como acceso al *Local Model* (*LM*).

Constructor

Podemos encontrar dos tipos de constructores. El primero no acepta atributos y hace una asignación rápida de los atributos obligatorios (*final*) y asigna *null* al *LM*. Este constructor se usa en la sección del *codex* únicamente para tener un objeto de la instancia *Ball* y poder resolver situaciones específicas.

El segundo constructor es el utilizado para realmente crear el objeto *Ball*, ya que permite asignar las coordenadas iniciales (x, y) y el *LM* a utilizar, además, internamente se asignan el resto de los atributos como *maxSpeed*, *diameter* y demás (ver *ilustración 5*).

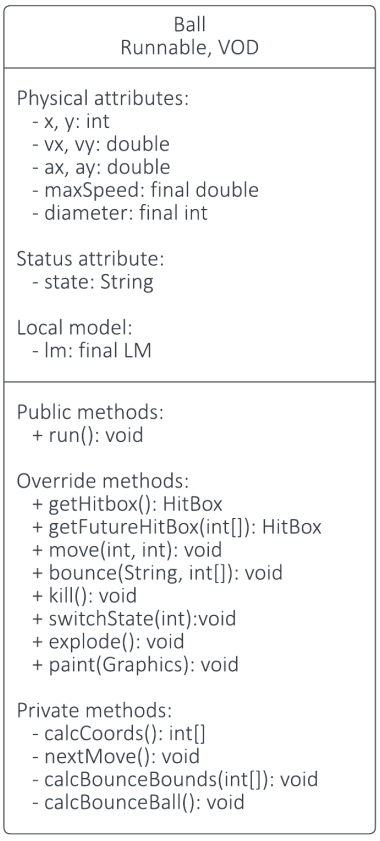


Ilustración 5. Ball en detalle.

Métodos

Override

Los métodos *override* provienen de la implementación de la interfaz *VOD* que establece la estructura básica de los objetos dinámicos del programa. A fecha de escritura del documento, estos métodos son:

GetHitbox

Sado pare extraer un objeto *HitBox* que se utilizará en el *Local model* (*LM*) para comprobar la colisión contra otros objetos.

```
@Override
    public HitBox getHitbox() {
        return new HitBox(x - (diameter/2), y - (diameter/2), x +
(diameter/2), y + (diameter/2));
    }
```

GetFutureHitBox

Idéntico a su homólogo, pero en vez de usar las coordenadas actuales del objeto usa las del siguiente movimiento.

```
@Override
    public HitBox getFutureHitbox(int[] coords){
        return new HitBox(coords[0] - (diameter/2), coords[1] -
(diameter/2), coords[0] + (diameter/2), coords[1] + (diameter/2));
    }
```

Move

Encargado de desplazar el objeto a las coordenadas recibidas por parámetro.

```
@Override
    public void move(int x, int y) {
        setX(x);
        setY(y);
    }
```

Bounce

Encargado de ejecutar los rebotes que pueda tener el objeto. Este método requiere como parámetros un *String* que se usa para decidir el tipo de rebote, y un set de coordenadas. En este momento existen dos posibilidades:

- *Ball*: Representa un rebote contra otro objeto *Ball* y se realiza mediante la ejecución de un método llamado *calcBounceBall()*.
- *Bounds*: Representa un rebote contra un objeto *Bound* que representa un borde de la ventana. Dado que es un caso con 8 posibles rebotes, este caso solo llama a otro método para calcular y realizar el rebote (*calcBounceBound()*).

```
@Override
    public void bounce(String action, int[] coords) {
        switch (action) {
            case Ball -> {
                calcBounceBall();

            }
            case Bounds -> {
                calcBounceBounds(coords);
            }
        }
    }
```

Kill

Este método no hace nada por el momento.

SwitchState

Método usado para cambiar el atributo de *state* del objeto. Requiere de un número entero como parámetro para realizar el cambio.

```
@Override
    public void switchState(int newState) {
        switch (newState) {
            case 0 -> state = alive;
            case 1 -> state = terminated;
        }
    }
```

Explode

Este método no hace nada por el momento.

Paint

Este método se encarga de la parte gráfica. Recibe como parámetro el objeto *Graphics* del *Viewer* lo que permite pintarse a sí mismo en la ventana. Dentro podemos encontrar dos tipos de polígonos, un rectángulo rojo que representa la *HitBox* del objeto, y el óvalo blanco que representa al propio objeto. La sección del código del rectángulo puede ser comentada o eliminada, su propósito es ayudar al programador.

```
@Override
    public void paint(Graphics g) {
        g.setColor(Color.RED);
        g.fillRect(x - diameter / 2, y - diameter / 2, diameter, diameter);

        g.setColor(Color.WHITE);
        g.fillOval(x - diameter / 2, y - diameter / 2, diameter, diameter);
    }
```

Públicos

El único método público actual es el *run()* encargado del hilo de ejecución del objeto (además, también es un método override). Este será el encargado de poner en funcionamiento los engranajes que mueven el objeto. Además, cuenta con una pausa entre movimientos de 16ms (cifra usada para aproximar a 60fps), y un *switchState()* para cambiar el estado a *alive*, de forma que el bucle podrá activarse. Mientras el estado no cambie a *termianted* este se ejecutará eternamente.

```
@Override
    public void run() {
        switchState(0);
        while (!state.equals(terminated)) {
            nextMove();
            try {
                Thread.sleep(16);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

Privados

Los métodos privados se usan internamente y no intervienen en ninguna actividad exterior al objeto. Podemos encontrar cuatro.

CalcCoords

Realiza el cálculo de las nuevas coordenadas y las devuelve como un array de enteros.

```
private int[] calcCoords() {
        int newX = (int) (x + vx);
        int newY = (int) (y + vy);

        return new int[]{newX, newY};
    }
```

NextMove

Le pide al *LM* si se puede desplazar y le pasa al método de este las nuevas coordenadas y a sí mismo para que el *LM* pueda tratar el caso.

```
private void nextMove() {
        lm.collideDetection(this, calcCoords());
    }
```

CalcBounceBall

Este método, que se utiliza al detectar una colisión con otro objeto tipo *Ball*, invierte los atributos de velocidad v_x y v_y y ajusta las coordenadas actuales.

```
 private void calcBounceBall() {
        vx = -vx;
        vy = -vy;

        setX((int)(x + vx));
        setY((int)(y + vy));
    }
```

CalcBounceBound

Este método se usa para calcular el rebote contra un objeto *Bound* y desplazar el objeto justo al borde para poder continuar con el siguiente movimiento, habiendo tratado el rebote.

```
private void calcBounceBounds(int[] coords) { // Cálculo del rebote con el
borde
        int coordX = coords[0];
        int coordY = coords[1];

        if (coordX - diameter <= 2) {
            vx = -vx;
            setX(2 + diameter);
        }
        if (coordX + diameter >= lm.getWidth() - 2) {
            vx = -vx;
            setX(lm.getWidth() - 2 - diameter);
        }

        if (coordY - diameter <= 2) {
            vy = -vy;
            setY(2 + diameter);
        }
        if (coordY + diameter >= lm.getHeight() - 2) {
            vy = -vy;
            setY(lm.getHeight() - 2 - diameter);
        }
    }
```

Cálculo de los bounce

El ajuste de las coordenadas que se puede ver en los métodos *bounce* de *Ball* se realizan por dos motivos. El primero es para que el bucle del *run()* pueda seguir operando y no se quede bloqueado en este punto, ya que si no se resuelve el conflicto con un movimiento se bloquea. Además, no se puede volver a ejecutar *nextMove()* ya que entra en un bucle de correcciones del cual no puede salir, dejando solo una hilera de errores infinita. El segundo motivo es uno matemático.

Por cómo opera el programa, las coordenadas actuales no son lo importante, lo que importa son las coordenadas futuras (*ilustración 6*). Partiendo de esta base, una colisión detectada se tendrá que resolver en el futuro del objeto, ya sea cambiando el vector velocidad ($\overrightarrow{v_{ij}}$) o la posición del objeto. Actualmente solo existen dos posibles casos, el choque contra un *Bound*, y el choque contra otra *Ball*. Vamos a tratar ambos casos para permitir una mejor compresión de los métodos *calcBounce()*.

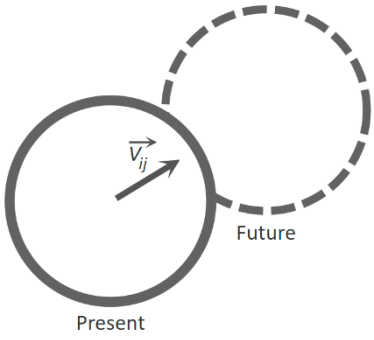


Ilustración 6. Cálculos de bounce.

Para saber si las coordenadas futuras (generadas por el *nextMove()*) del objeto van a colisionar contra algo, el *LM* pide a cada objeto su *HitBox* (rectángulo rojo en la ilustración 7, azul en el caso del *Bound*) y comprueba si estos intersecan. En caso de no producirse una intersección, no hay colisión, pero si en efecto intersecan, se debe resolver el movimiento de una forma específica.

Si la colisión de *Ball* ha sido con otra igual, el proceso es simple, actualizamos el vector velocidad ($\overrightarrow{v_{ij}}$) mediante una inversión de los atributos v_x y v_y y ajustamos las coordenadas (x, y) actuales a $n + v_n$ (siendo n la generalización de x e y). De esta forma creamos un movimiento natural que provoca la finalización de toda la cadena de eventos explicado en el apartado *Proceso de vida de un objeto* e inhabilita la entrada en el bucle infinito de errores mencionado con anterioridad.

La otra posibilidad de colisión es con un *Bound*. Este caso es más complejo, ya que la forma de rebotar dependerá de cual de los bordes hemos golpeado, haciendo que haya cuatro casos posibles, *N*, *S*, *E* y *W*. En esencia, lo que podemos comprobar es que podemos dividir los bordes en función de la velocidad que le afecta. En más detalle, como el control del movimiento viene dado por el vector velocidad ($\overrightarrow{v_{ij}}$) y este se puede descomponer en los atributos del objeto v_x y v_y, podemos asignar los bordes que cambian cada componente, i.e. el borde que se encuentra en la zona superior de la pantalla *N* solo podrá cambiar la velocidad vertical (v_y) del *Ball* que colisione contra él, ya que no tendría sentido que si te das contra el techo este te deflecte hacia la izquierda o derecha pero no evite que te sigas dando contra él. De esta forma y mediante esta analogía se pueden separar bordes en función del componente que cambia, dejándonos los siguientes grupos:

$$v_X \rightarrow E\ y\ W$$

$$v_y \rightarrow N\ y\ S$$

Volviendo al ciclo de vida de *Ball*, una vez detectado la colisión contra un *Bound*, y sabiendo que cada uno de ellos afecta a una de las componentes de $\overrightarrow{v_{ij}}$, podemos realizar los cálculos necesarios y trata cada caso. Para ello usamos dos comprobaciones que son extrapolables a cada eje:

$$newN - \emptyset \leq 2$$

$$newN + \emptyset \geq Dimension - 2$$

Siendo *newN* la generalización de las coordenadas calculadas en *calcCoords()*, $\emptyset$ el diámetro y *Dimension* la altura o el ancho de la ventana. La existencia del número 2 en las fórmulas se explica en el apartado de *Inicio de la aplicación* y con más detalle en *HitBox*.

Una vez entendido esto, la primera fórmula codifica los casos *N* y *W*, y la segunda *S* y *E*. Usando el ejemplo de la imagen a la izquierda, tenemos la situación *A*, donde *LM* ha detectado nuestra futura colisión con un *Bound*, concretamente el que se encuentra en el este (*E*). A partir de esto podemos ver que si mantenemos el rumbo no solo impactaremos, sino que entraremos dentro del *Bound*, de modo que tenemos que evitar que ocurra.

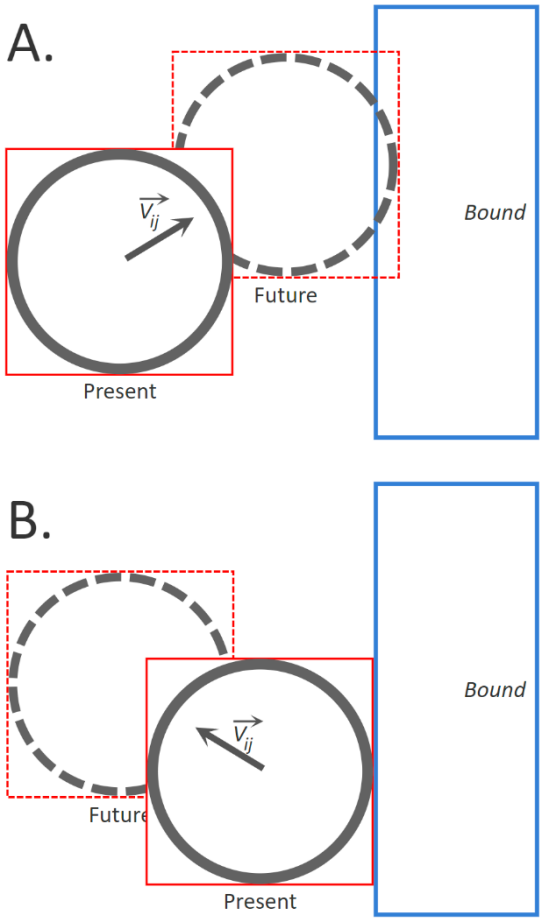


Ilustración 7. Hitboxes y tratamiento de Bound.

Esto se consigue descomponiendo $\overrightarrow{v_{ij}}$ en v_x y v_y, no nos interesa tanto sus valores, como sus signos. En este caso nos estamos desplazando en dirección *NW* (ver la flecha negra en la circunferencia del *presente*), de modo que sabemos que $v_x > 0$ y que $v_y < 0$, o, dicho de otra forma, v_x es positiva y v_y es negativa, esto es de interés pues, sin necesidad de tener un apoyo gráfico (como es el caso) podemos descartar impactos contra *S* y *W*. Todo este proceso está condensado en el *calcCoords()* que nos da un set de coordenadas en base a estas velocidades, por lo que comparando estas en las fórmulas anteriores podemos determinar que borde impactamos. Volviendo al caso que estábamos analizando, vamos a solucionar la componente x. Sabemos por *calcCoords()* que $newX = x + v_x$ y que esto nos lleva a impactar el *Bound*, por lo que si ponemos esta coordenada en las fórmulas anteriores obtendremos una situación, digamos que $x = 10$, que $\emptyset = 5$, que $v_x = 15$ y que el ancho de la ventana es 25:

Caso 1:

$$\begin{matrix} newX = x + v_x \\ newX - \emptyset \leq 2 \end{matrix} \xrightarrow{Lleva\ a} x + v_x - \emptyset \leq 2 \xrightarrow{Sustituimos} 10 + 15 - 5 \leq 2 \xrightarrow{Operamos} 20 \leq 2 \rightarrow False$$

Caso 2:

$$\begin{matrix} newX = x + v_x \\ newX + \emptyset \geq Ancho - 2 \end{matrix} \xrightarrow{Lleva\ a} x + v_x + \emptyset \geq 25 - 2 \xrightarrow{Sustituimos} 10 + 15 + 5 \geq 25 - 2 \xrightarrow{Operamos} 30 \geq 23 \rightarrow True$$

Con esta aplicación de las fórmulas podemos comprobar que verdaderamente vamos a impactar e incluso salir del *Bound* y contra cual lo vamos a hacer, de forma que lo que tenemos que hacer es evitar esto, pero para incrementar el realismo de la aplicación vamos a mover la *Ball* hasta el límite posible que se puede observar en la parte *B* de la *ilustración 7*. Para ello invertimos la polaridad de v_x mediante $v_x = -v_x$ y ajustamos nuestra x actual de forma que estemos en el límite posible, esto se realiza con $setX(2 - \emptyset)$ si el caso 1 da verdadero o con $setX(Width - 2 - \emptyset)$ si el caso 2 es verdadero (estas fórmulas nos devuelven el punto exacto donde estamos rozando el *Bound* como se aprecia en la *ilustración 7*). Con esto ya está todo listo, faltaría solo resolver lo mismo para la componente vertical y y habríamos acabado. Tras esto el programa detecta que hemos resuelto la colisión y que nos hemos movido, de forma que evitamos bucles de errores y comprobaciones y podemos ejecutar el siguiente *nextMove()*.

Bound

Constituido principalmente por dos sets de coordenadas (x_1, y_1) y (x_2, y_2), representa el objeto que delimita los bordes de la ventana, encerrando a todos los *objetos dinámicos* (*VOD*) entre ellos.

Al igual que los objetos *Ball*, tiene un constructor vacío dedicado a la parte del *Códex* y otro a crear el objeto en sí. Además, al ser un objeto que implementa *VOD* tiene todos los métodos *override*, pero el único de estos a tener en cuenta es el *getHitbox()* ya que, junto al *paint()* (cuando se requiere de él, como es el caso al realizar pruebas), es el único que se va a usar durante el ciclo de vida de la aplicación.

```
@Override
    public HitBox getHitbox() {
        return new HitBox(x1, y1, x2,
y2);
    }
```

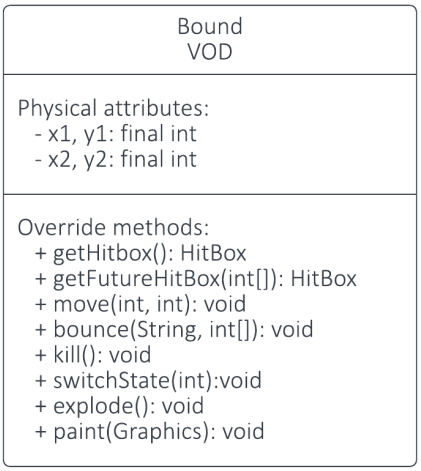


Ilustración 8. Bound en detalle.

HitBox

Esta clase se utiliza para comprobar las colisiones entre objetos. Recibe como parámetros en el constructor cuatro números enteros que representan las coordenadas de la esquina superior izquierda e inferior derecha.

Es importante remarcar que, a pesar de que el constructor recibe dos sets de coordenadas, estos no constituyen los atributos del objeto. Como se puede observar, el tercer y cuarto parámetro son una resta de componentes. Esto se debe a que el objeto *Rectangle* que extiende *HitBox* necesita como parámetros la esquina superior izquierda, el ancho y el alto, de forma que en el *super()* se debe seguir este procedimiento. Como sabemos que:

$$Width = x_f - x_i$$
$$Height = y_f - y_i$$

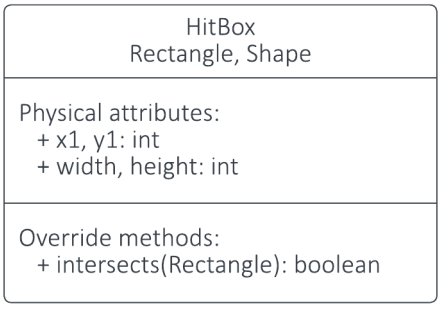


Ilustración 9. HitBox en detalle.

Siendo $f \rightarrow final\ (2)$ e $i \rightarrow inicial\ (1)$. De esta forma, conociendo los puntos remarcados en la *ilustración 10* podemos construir cualquier tipo de rectángulo. Se debe remarcar la importancia de pasarle al constructor de HitBox primero la esquina superior izquierda y luego la inferior derecha. Si no se procede de esta forma se producirán errores más adelante.

```
public HitBox(int x1, int y1, int x2, int y2) {
        super(x1, y1, x2 - x1, y2 - y1);
    }
```

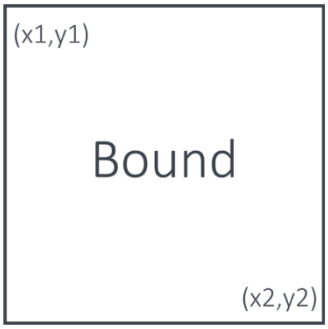


Ilustración 10. Puntos necesarios del rectángulo.

Por cómo funciona el Rectangle, se podría crear un rectángulo de dimensión uno, es decir, que solo tiene una componente (transversal, x, o vertical, y), diferente. Es decir, una de las componentes será la misma en ambos sets de coordenadas. Esto provoca que, en vez de ser un rectángulo, sea una línea, lo que genera ciertas consideraciones para tener en cuenta. A pesar de que se pueda resolver la intersección entre una recta (la línea) y un plano (el rectángulo), su resolución es mucho más compleja que ver una intersección entre dos planos (dos rectángulos). A raiz de esto, el método *intersetcs()* no está preparado para manejar este caso, pues está preparado para dos planos.

Por este motivo, todos los objetos deben seguir lo mencionado con anterioridad y proporcionar al constructor de HitBox dos puntos diferentes en ambas componentes de forma que ninguna de las x e y se pueden repetir.

Local Model – LM

Esta clase se encarga del procesamiento de solicitudes de movimiento de los diferentes *VOD*, del manejo de colisiones y de la ejecución de instrucciones.

Atributos

El *LM* se compone del alto y ancho de la ventana de la aplicación, información recibida a través del *Local Controller* (*LC*), de un *ArrayList* que usa *VOD* como componente genérica para poder guardar todos los *objetos dinámicos* en una misma lista, y el propio *LC*.

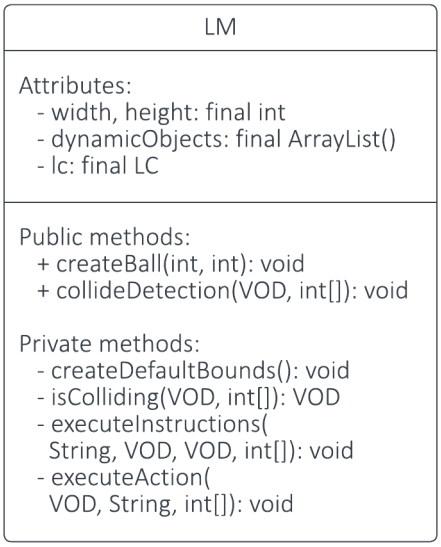


Ilustración 11. LM en detalle.

Constructor

El *LM* solo tiene un constructor y recibe como parámetros el ancho y alto, y el *LC*. Dentro, asigna los valores a los atributos correspondientes, crea el *ArrayList* vacío, y ejecuta *createDefaultBounds()*, que se explica más adelante.

Métodos

Públicos

CreateBall

Método encargado de crear un objeto tipo *Ball*. Recibe como parámetros dos números enteros (*int*), x e y, que usa a la hora de instanciar una nueva *Ball*. Una vez instanciado, y al ser un tipo de *Runnable*, crea un objeto tipo *Thread* y mediante el comando *Thread.start()* inicia el *run()* de la nueva *Ball*.

```
public void createBall(int x, int y) {

        // Creación de una bola:
        Ball ball = new Ball(x, y, this);
        dynamicObjects.add(ball);

        // Inicio de su thread:
        Thread thread = new Thread(ball);
        thread.start();
    }
```

CollideDetection

Este método se encarga de iniciar la comprobación de colisiones. Recibe por parámetros un objeto *VOD* que es el propio solicitante (es decir, si una *Ball* llama a este método se pasa a sí misma con un *this*, ver método en apartado *nextMove()* de *Ball*), y un *int[]* que contiene las nuevas coordenadas del solicitante.

Una vez recibos los parámetros crea un objeto *VOD*, de mismo nombre, y llama al método *isColliding()*, que se explica más adelante. Este método puede o no devolver un objeto. Continua mediante una condición, si la variable *vod* (referencia a la variable, no tipo de objeto, de ahí las minúsculas) está vacía, significa que no se ha detectado ninguna posición, por lo que *LM* ejecuta el método *move()* de *Ball* y finaliza.

Pero en caso de que no esté vacía, significa que se ha detectado una colisión y que se ha almacenado en esta variable el objeto contra el que el solicitante ha impactado. En este momento se pone en marcha una cadena de llamadas (comentado en más profundidad en el apartado del *Códex*) que acaba por recibir un código en formato *String* que contiene el proceder de la situación, ya que el *LM* no resuelve la solución de por sí. *LM* llama al *LC* y este a su vez extrae del *Códex* la resolución de la situación, esta es devuelta con la forma de proceder.

El *String* con el procedimiento se guarda en una variable y se pasa como parámetro a *executeInstructions()* junto a los dos objetos (solicitante y colisionado) que intervienen y el set de coordenadas (*int[]*) que provoca el conflicto. Hecho esto el método finaliza.

```
    public void collideDetection(VOD object, int[] coords) {
        VOD vod = isColliding(object, coords);

        if (vod != null) {
            String actions = lc.solveSituation(object, vod);
            executeInstructions(actions, object, vod, coords);
        } else {
            object.move(coords[0], coords[1]);
        }
    }
```

Privados

CreateDefaultBounds

No recibe ningún parámetro y se encarga de crear los cuatro objetos *Bound* que rodearán el área de juego. Por los motivos explicados en el apartado de *HitBox*, no podemos crear este objeto de forma que componga una línea, de modo que cada *Bound* deberá ser bidimensional, y deberá establecer la primera coordenada de este de forma que sea siempre la correspondiente a la esquina superior izquierda, y la segunda, la inferior derecha. En este caso se ha decidido de forma arbitraría usar el número dos, de forma que lo usado para crear los *Bounds* será el 0, 2, el ancho, y el alto, lo que explica lo que se puede ver en el método.

```
private void createDefaultBounds() {
        Bound N = new Bound(0, 0, width, 2);
        Bound S = new Bound(0, height - 2, width, height);
        Bound E = new Bound(width - 2, 0, width, height);
        Bound W = new Bound(0, 0, 2, height);

        dynamicObjects.add(N);
        dynamicObjects.add(S);
        dynamicObjects.add(E);
        dynamicObjects.add(W);
    }
```

IsColliding

El método recibe como parámetros un *VOD* y un set de coordenadas (*int[]*). Luego, procede a iterar mediante un bucle *for-each* todos los elementos guardados en *dynamicObjects* (*ArrayList*). Primeramente, evita que se analice a si mismo mediante $vod \neq object$, luego, por cada objeto en la lista extrae la *HitBox* y lanza el método *intersects()* para ver si la *HitBox* de ambos objetos intersecan en las futuras coordenadas. En caso de no intersecar, retorna *null* y finaliza. Pero si intersecan devuelve el objeto contra el que va a colisionar.

```
private VOD isColliding(VOD object, int[] coords) {
        for (VOD vod : dynamicObjects) {
            if (vod != object) {
                if
(object.getFutureHitbox(coords).intersects(vod.getHitbox())) {
                    return vod;
                }
            }
        }
        return null;
    }
```

ExecuteInstructions

Con este método entramos verdaderamente en el terreno de la ejecución del *Códex* y la solución de interacciones entre *VODs*. Advertimos paciencia, pues el cómo funcionan los códigos (*String*) se explica en el apartado propio del *Códex*. De momento las explicaciones se centrarán en la parte que maneja el *LM*.

Este método recibe como parámetros un *String* con las acciones para cada objeto que interviene en la colisión, los dos objetos que intervienen, y el set de coordenadas (*int[]*). Si se llama a este método es muy probable que se haya producido una colisión porque ha sido llamado después de la detección de una, pero por redundancia y seguridad se inicia con un condicional que evalúa si el *String* verdaderamente existe y tiene contenido. En caso de que verdaderamente esté vacío deja de proceder. Si verdaderamente el método ha recibido las instrucciones procede a guardar cada *VOD* en un diccionario donde se le asigna una clave para poder identificarlos (más información en *Códex*), luego procede a pasarle a *executeAction()* cada objeto que interviene y la resolución para ese objeto concreto. En este caso vuelve a encontrarse una redundancia de seguridad.

Si por algún casual se llegan a activar las redundancias, es decir, ha sucedido un error, o bug, o fallo por modificaciones futuras, los objetos se quedarán detenidos tanto gráficamente como algorítmicamente, pues de momento no tienen tratamiento. De forma que si durante alguna ejecución dos objetos se quedan completamente estáticos significa que ha habido un fallo inesperado, pero lo que se recomienda dar parte.

```
private void executeInstructions(String actions, VOD objectA, VOD objectB,
int[] coords) {
        if (actions != null) {
            Map<String, VOD> objects = new HashMap<>();
            objects.put("objectA", objectA);
            objects.put("objectB", objectB);

            String[] individualActions = actions.split("/");

            for (String action : individualActions) {

                String[] parts = action.split(":");
                VOD object = objects.get(parts[0]);
                String objectAction = parts[1];
                if (object != null) {
                    executeAction(object, objectAction, coords);
                }
            }
        }
    }
```

ExecuteAciton

Este método recibe como parámetros un objeto *VOD*, un *String* que contiene la acción para el *VOD*, y el set de coordenadas generado con anterioridad por el *Ball*. El método evalúa la acción contenida en el *String* y en función de cada caso ejecuta una serie de métodos. El único en el que *LM* interviene de forma externa al *VOD* es en el caso *terminate*, donde se elimina al *VOD* del *ArrayList*.

```
private void executeAction(VOD object, String action, int[] coords) {
        switch (action) {
            case "bounce(Ball)" -> object.bounce("Ball", coords);
            case "bounce(Bound)" -> object.bounce("Bounds", coords);
            case "terminate()" -> {
                object.switchState(1);
                dynamicObjects.remove(object);
            }
```

```
        }
    }
```

Local controller – LC

Esta clase conforma el centro neurálgico del apartado algorítmico y visual, pues conforma el centro de la relación que mantienen la parte visual (*Local Viewer* y *Viewer*), el *Local Model* y el *Códex*.

Atributos

Los atributos del *LC* son el ancho y alto de la ventana, el *Local model*, el *Local Viewer* y el *Códex*.

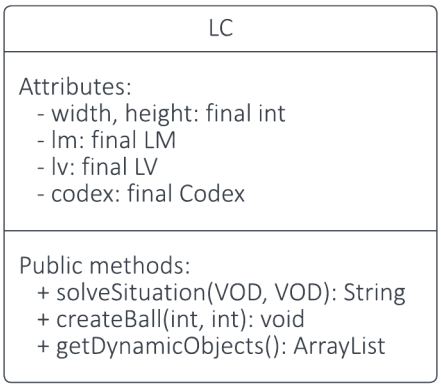


Ilustración 12. LC en detalle.

Constructor

El constructor del Local Controller recibe por parámetros el dos números enteros para configurar el tamaño de la ventana de la aplicación. Una vez asignados a los atributos correspondientes, crea tanto el *Local model*, como el Lo*cal Viewer*, y el *Códex*.

Métodos

Públicos

SolveSituation

Este método recibe como parámetros dos *VODs* y los envía al Códex para que analice y retorne el *String* con la solución. Una vez recibido, lo devuelve al solicitante.

```
public String solveSituation(VOD objctA, VOD objctB) {
        return codex.getRule(objctA, objctB);
    }
```

CreateBall

Recibe como parámetros dos números enteros (*int*), x e y, que envía al *LM* para que ejecute el mismo método.

```
public void createBall(int x, int y) {
        lm.createBall(x, y);
    }
```

GetDynamicObjects

No recibe parámetros y devuelve el *ArrayList* del *LM*.

```
public ArrayList<VOD> getDynamicObjects() {
        return lm.getDynamicObjects();
    }
```

Local Viewer – LV

El LV constituye la primera pieza del apartado visual y se encarga del manejo de la ventana de la aplicación (*Frame*). Nos vamos a detener en este punto para explicar de que se compone, simplificadamente, una ventana en Java. Una ventana se puede dividir en diferentes capas, cada una en una profundidad diferente (ver ilustración. El frame constituye el nivel mas bajo posible, por delante de este se irán estructurando diferentes paneles o capas, cada uno con un contenido, en función de lo que se quiera mostrar visualmente. Usando una analogía, podemos imaginar un carrusel de imágenes, en este carrusel podemos ir viendo diferentes imágenes, pero el carrusel per se no cambia. En este caso, el carrusel sería el frame y las imágenes los diferentes paneles.

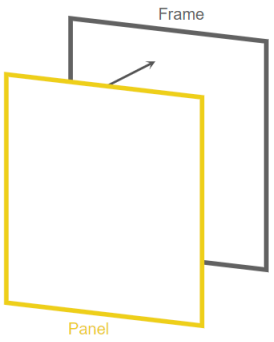


Ilustración 13. Capas de ventana.

Atributos

El *LV* solo tiene dos atributos, el *LC* necesario para poder comunicarse con el resto de la aplicación, y el *Viewer*.

Constructor

El constructor recibe como parámetros el *LC*, el ancho y el alto de la aplicación. Dentro, ejecuta un método llamado *setUI()*, crea el Viewer necesario para acabar de iniciar la parte visual y lo añade al frame (es decir, a si mismo para tener un panel). Hecho esto procede a ejecutar *pack()* y *setResizable(false)* que ajusta las dimensiones de la ventana al contenido y evita el cambiar el tamaño de esta respectivamente, y para finalizar arranca el hilo de ejecución del *Viewer*.

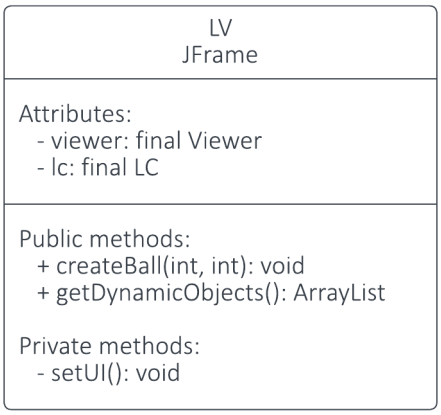


Ilustración 14. LV en detalle.

Métodos

Públicos

CreateBall

Método que recibe como parámetros dos números enteros (*int*), x e y, proporcionados por el *Viewer*, y los envía al *LC*.

```
public void createBall(int x, int y) {
        lc.createBall(x,y);
    }
```

GetDynamicObjects

No recibe parámetros y devuelve el *ArrayList* del *LC*.

```
public ArrayList<VOD> getDynamicObjects() {
        return lc.getDynamicObjects();
    }
```

privados

setUI

Este constituye el único método privado del *LV* y se encarga de configurar los ajustes del frame.

```
private void setUI() {
        setTitle("Bouncing ball");
        setLocation(660, 240);
        setResizable(false);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```

Viewer

El *Viewer* constituye el panel donde se irán dibujando y actualizando los objetos visualmente, además cuenta con la capacidad de crear un objeto Ball al hacer clic izquierdo con el ratón en las coordenadas donde se ha realizado el clic.

Atributos

Encontramos el *Local Viewer*, el ancho y alto de la ventana, que será necesario para realizar las operaciones del panel, y un objeto llamado *BufferStrategy*.

Constructor

El constructor recibe como parámetros la instancia del *LV*, el ancho y el alto. Inicializa la variable del *BufferStrategy* como *null* para mas adelante poder utilizarlo, luego procede a llamar a *setUI()*. Por último, encontramos un objeto *Listener*, que se encarga de estar atento a cuando se pulsa el botón izquierdo del ratón para ejecutar el *createBall()* en las coordenadas del clic.

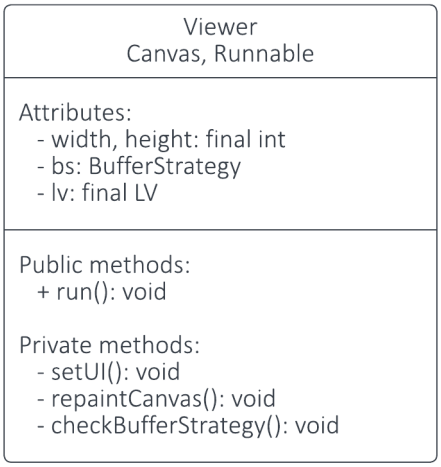


Ilustración 15. Viewer en detalle.

Métodos

Públicos

Run

El único método público que existe en el *Viewer* es el *run()*, que se encarga de ir llamando a *repaintCanvas()* en cada iteración del bucle. Este contiene una pausa entre iteraciones de 16ms para aproximarse a los 60 fps.

$$Delay = \frac{1000}{Desired\ FPS} \rightarrow Delay = \frac{1000}{60} \approx 16$$

El 1000 se obtiene debido a que el *Thread.sleep()* opera con milisegundos, por lo que este sirve a modo de conversión a segundos.

```
@Override
    public void run() {
        while (true) {
            repaintCanvas();

            try {
                Thread.sleep(16);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
```

Privados

SetUI

Este método se encarga de ajustar la configuración del panel como el tamaño y el color de fondo.

```
private void setUI() {
        setPreferredSize(new Dimension(width, height));
        setBackground(new Color(30, 30, 30));
    }
```

RepaintCanvas

Este método se encarga de ir actualizando el contenido del panel, pintando así la actualización del apartado gráfico. Inicia obteniendo el objeto Graphics necesario para crear objetos bidimensionales de forma gráfica, luego crea un nuevo frame (no confundir con el frame del LV, este se refiere a uno de los 60FPS que debe crear al segundo) e inicia el bucle recorriendo la lista de objetos actuales que se encuentra en el *Local Model*, *LM*. Por cada objeto llama a su método *paint()* para representarlo en el frame y pasa al siguiente hasta haberlos representado a todos. Luego el *run()* procede a repetir el proceso.

```
private void repaintCanvas() {
        checkBufferStrategy();
        Graphics g = bs.getDrawGraphics();

        g.clearRect(0, 0, width, height);

        for (VOD dynamicObject : lv.getDynamicObjects()) {
            dynamicObject.paint(g);
        }

        bs.show();
        g.dispose();
    }
```

Códex

A fecha de redacción de es te manual, el Códex puede resultar la parte más confusa de todo el programa. La idea detrás del funcionamiento es que reciba una situación donde se haya producido una colisión entre dos *VODs* y enviar la resolución de este problema con un código en *String* (ver *Situation*).

Atributos

El único atributo es un *Map()* llamado codex que guarda pares clave-valor. Este guarda cada situación que puede ocurrir con los *VODs* y como un objeto que contiene la resolución de dicha situación.

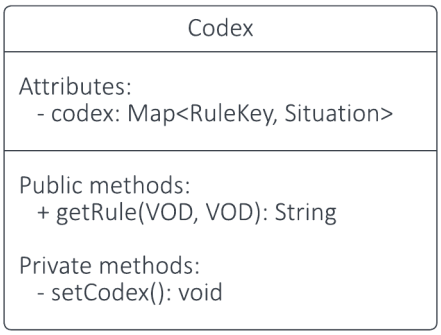


Ilustración 16. Códex en detalle.

Constructor

Este no tiene parámetros, lo único que realiza es la creación del *Map()* mediante un *new HashMap()* y una llamada al método *setCodex()*.

Métodos

Públicos

GetRule

Como único método público del *Códex*, este recibe como parámetro dos *VODs* y crea una nueva clave mediante la creación de una *RuleKey*, a la cual le pasa ambos *VODs*. Gracias a la nueva *RuleKey* podemos mirar en el *Map()* y extraer la solución (*String*) para devolvérsela al *LC*.

Mirando con un poco mas en profundidad, y saltada la redundancia de seguridad del *if*, lo que ocurre es que, del objeto que nos devuelve el *Map()* podemos extraer el método donde se guardan las acciones a realizar por cada *VOD*. Este método extraído se guarda en una variable, la cual se puede ejecutar para obtener, en este caso, el *String* que codifica las acciones a tomar.

```
public String getRule(VOD objctA, VOD objctB) {
        RuleKey key = new RuleKey(objctA, objctB);

        Situation rule = codex.get(key);

        if (rule != null) {
            BiFunction<?, ?, String> function = rule.getFunction();
            return ((BiFunction<VOD, VOD, String>) function).apply(objctA,
objctB);
        }
        return null;
    }
```

Privado

SetCodex

Este método crea las entradas del codex. Usando una analogía, añade las páginas del reglamento.

```
private void setCodex() {
        codex.put(new RuleKey(new Ball(), new Ball()), new Ball_Ball());
        codex.put(new RuleKey(new Ball(), new Bound()), new Ball_Bound());
    }
```

RuleKey

Este objeto se encarga de guardar pares de VOD que representan una situación concreta que se puede dar en el programa. Un ejemplo de esto es una situación donde intervengan dos *Ball* o un caso de *Ball-Bound*. Cada *RuleKey* está diseñada para ser usada como clave en el *Map()* del Códex.

Atributos

Cada *RuleKey* tiene como atributos dos VOD llamados *objectA* y *objectB*.

Constructor

El constructor recibe por parámetros dos VOD y los asigna a las variables correspondiente, definiendo así la clase.

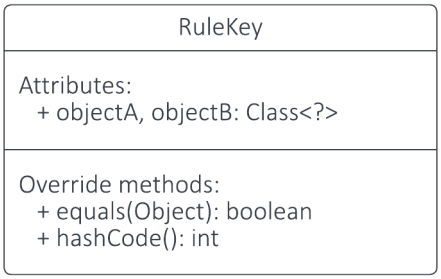


Ilustración 17. RuleKey en detalle.

Explicando en más profundidad. Como tal no se pueden definir los atributos como VODs, ya que esto provocaría la pérdida del tipo de objeto que era con anterioridad. Todos los *dynamic object* implementan la interfaz VOD, pero VOD no es convertible a ningún otro objeto. Explicado con una analogía, es una vía de único sentido, puedes pasar de *dynamic object* a *VOD*, pero no a la inversa. Esto produce que el tipo de dato no pueda ser determinado en la programación de RuleKey, de ahí que se va lo siguiente:

```
public Class<?> objectA;
public Class<?> objectB;
```

Para poder resolver esta situación el constructor guarda el objeto y su clase:

```
public RuleKey(VOD objectA, VOD objectB) {
        this.objectA = objectA.getClass();
        this.objectB = objectB.getClass();
    }
```

Métodos

Override

Equals y hashCode

Al ser *RuleKey* una clase que va a ser usada en un *Map()*, requiere sí o sí de los métodos *equals()* y *hashCode()*. Empezando por *equals(Object o)*, compara un objeto consigo mismo, para ello comprueba si la instancia del objeto es idéntica a la *RuleKey* actual, cosa que solo ocurre si sus atributos son de la misma clase, es decir, una *RuleKey(Ball,Ball)* nunca dirá que es igual a *RuleKey(Ball,Bound)*. Gracias a esto se tiene una comparación de claves. El método *equals()* únicamente devuelve el *hash* (código único) de la *RuleKey* en la que se ejecute.

```
@Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        RuleKey ruleKey = (RuleKey) o;
        return objectA.equals(ruleKey.objectA) &&
objectB.equals(ruleKey.objectB);
    }
    @Override
    public int hashCode() { // Return de los objetos
        return Objects.hash(objectA, objectB);
    }
```

Situation

Esta es una interfaz que será usada por cada entrada del *Códex*. Su único componente es un método llamado *getFunction()*, que devuelve una función que acepta como entrada dos objetos cuales sean y devuelve un *String*.

Este *String* sigue el siguiente patrón ":/:", donde la resolución de cada objeto está separada por /, y el objeto y su acción se separan por :. Para un ejemplo práctico ver *rule()* de *BallBall* o *BallBound*.

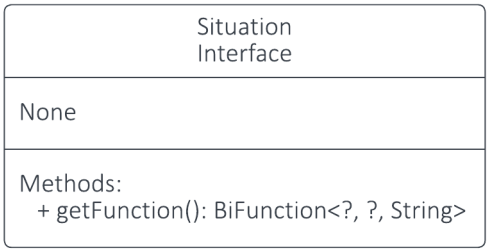


Ilustración 18. Situation en detalle.

BallBall

Clase utilizada para guardar el proceder en la situación *Ball-Ball*. Su funcionamiento es simple, cuando el método de *getFunction()* sea llamado desde el *Códex*, este retornará el resultado de ejecutar *getRule()*, que devuelve un *String* con el código para resolver esta situación en específico.

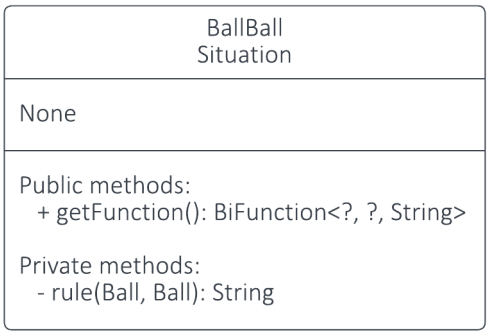


Ilustración 19. BallBall en detalle.

```
    @Override
    public BiFunction<Ball, Ball, String> getFunction() {
        return this::rule;
    }

    // Private methods:
    private String rule(Ball ball1, Ball ball2) {
        return "objectA:bounce(Ball)/objectB:bounce(Ball)";
    }
```

BallBound

Clase utilizada para guardar el proceder en la situación Ball-Bound. Su funcionamiento es simple, cuando el método de getFunction() sea llamado desde el Códex, este retornará el resultado de ejecutar getRule(), que devuelve un String con el código para resolver esta situación en específico.

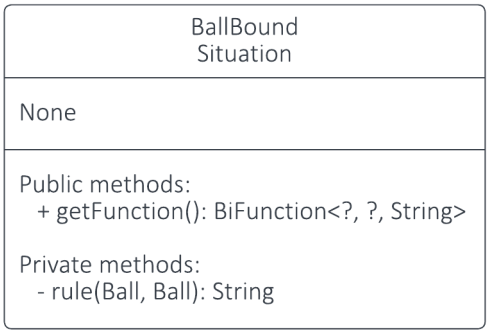
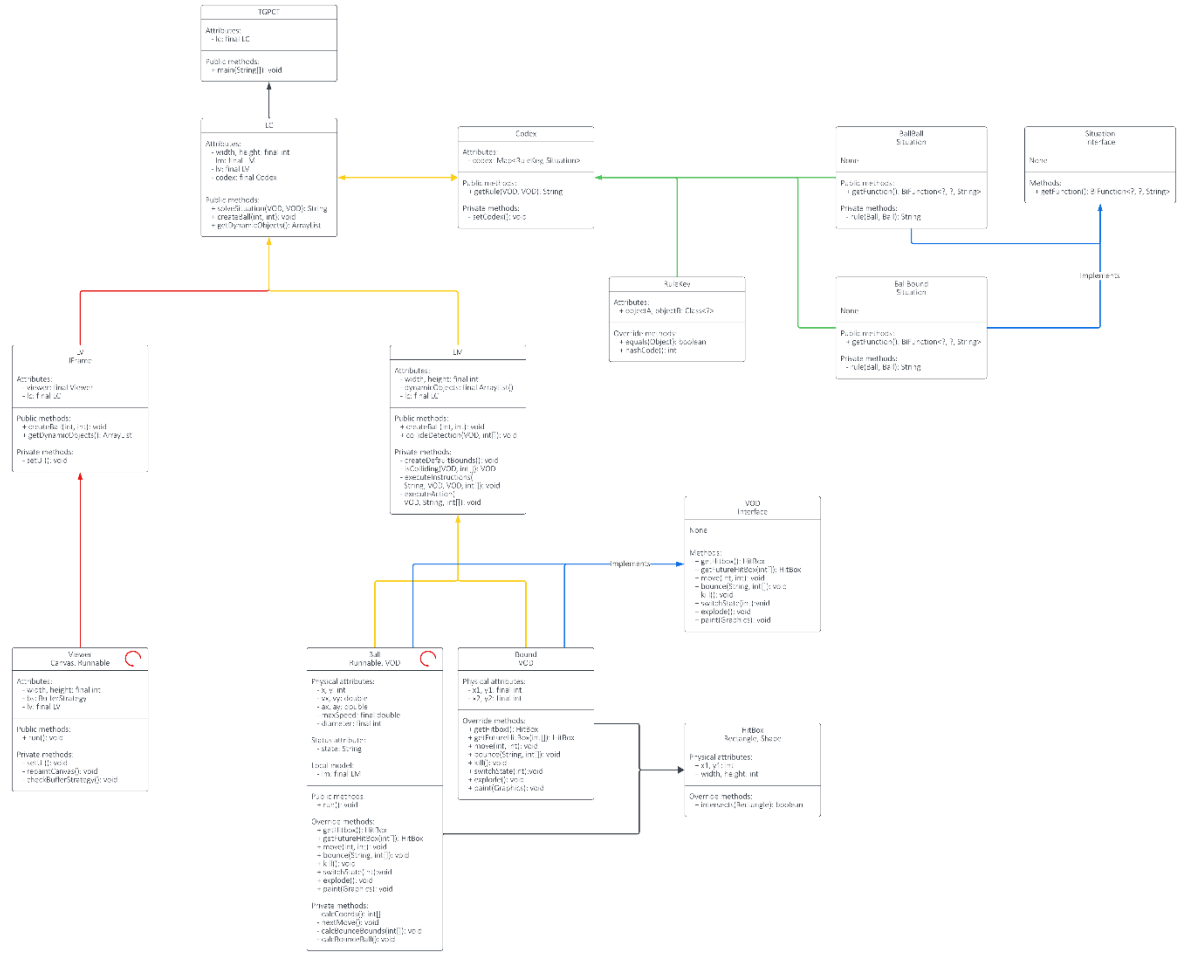


Ilustración 20. BallBound en detalle.

```
private String rule(Ball ball, Bound bound) {
        // Lógica específica para Ball-Bound
        return "objectA:bounce(Bound)/objectB:nothing()";
    }
```

Anexos

UML detallado



Enlace al UML