MEMORIA DE LA ASIGNATURA DSC

PONG

DAVID GÓMEZ TORRECILLA

ÍNDICE DE LA MEMORIA

1. DESCRIPCIÓN DEL PROYECTO

El proyecto se basa en el desarrollo del conocido juego PONG. El juego consiste en una simulación del tenis de mesa (ping-pong) dónde el jugador debe conseguir que la pelota pase la paleta del contrincante.

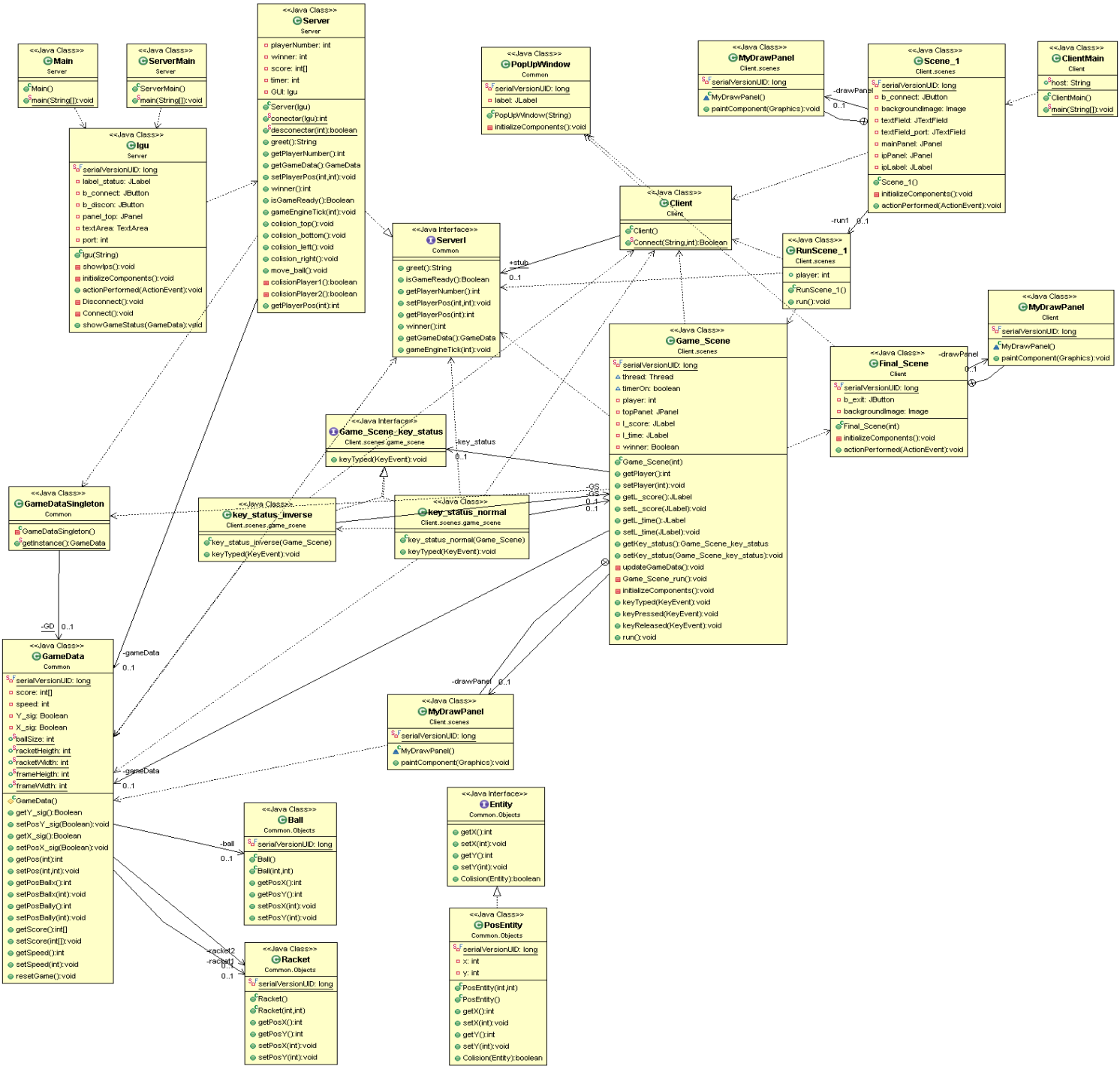
Para hacer el proyecto, hemos optado por diseñar desde cero el juego, aprovechando la realización del proyecto en otra asignatura. Para dar soporte multijugador, hemos optado por la tecnología RMI (Remote Method Invocation) de Java, puesto que era el requisito de la otra asignatura. Como IDE de programación hemos optado por Eclipse.

Tras conectar el servidor de juego, procedemos a arrancar cada uno de los clientes. En la pantalla inicial del cliente, se nos solicita la ip y el puerto del servidor al que conectarnos. Una vez conectados 2 jugadores al servidor, iniciamos la partida. Para jugar, presionaremos la tecla “Q” para elevar la paleta y “A” para descenderla.

Si presionamos la tecla “I”, el orden de las teclas cambiará, por lo que levantaremos la paleta con la “A” y la descenderemos con la “Q”. Y así sucesivamente.

2. DIAGRAMA DE CLASES

En el diagrama de clases, podemos observar como la aplicación se divide en Cliente-Servidor, utilizando los objetos GameData para el intercambio de información entre cliente-servidor.



3. PATRONES DE DISEÑO APLICADOS

Patrón SINGLETON: Instancia única de partida

Una de las restricciones que hemos puesto, es que cada servidor sólo pueda jugar una partida de forma simultánea. Es por ello, que hemos implementado este patrón en el objeto que encapsula todos los datos de la partida (objeto GameData). Así nos aseguramos que no existen dos instancias de partidas.

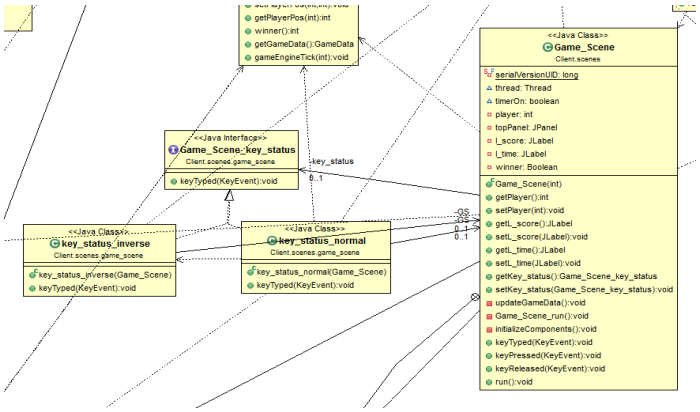
```
public class GameData implements Serializable{
    /**
     *
     */
    private static final long serialVersionUID = 1L;

    private Racket racket1;
    private Racket racket2;
    private Ball ball;

    private int[] score;
    private int speed;
    private Boolean Y_sig,X_sig;

    public static int ballSize=20;
    public static int racketHeigth=75;
    public static int racketWidth=20;
    public static int frameHeigth=550;
    public static int frameWidth=800;

    protected GameData(){
        racket1=new Racket(10,(frameHeigth/2+racketHeigth/2)-100);
        racket2=new Racket(frameWidth-30,(frameHeigth/2+racketHeigth/2)-100);
        ball=new Ball();

        score=new int[2];
        score[0]=0; score[1]=0;
        speed=5;
        X_sig=true;
        Y_sig=true;
    }

    public Boolean getY_sig() {
```

Patrón DECORADOR: Intercambio de teclas.

En el juego, si el jugador presiona la tecla “i”, el comportamiento de las teclas de juego se intercambia. Para ello hemos elegido el patrón puente. Dicho patrón nos permite el cambio de manejador de teclado de forma dinámica en el cliente.

La clase GameScene tiene un objeto de tipo estático `Game_Scene_key_status`, en el que delega el manejo de las pulsaciones del teclado. Así pues, puede tomar en tiempo de ejecución dos objetos distintos, intercambiables presionando la tecla “i”. Es por ello que podemos cambiar el comportamiento de las teclas en tiempo de ejecución.

Podemos ver como, según cuenta el patrón, el comportamiento de la función keyTyped cambía según el contenido dinámico del decorador “Game_Scene_key_status”.



➔ Key_status_inverse implements Game_Scene_key_status

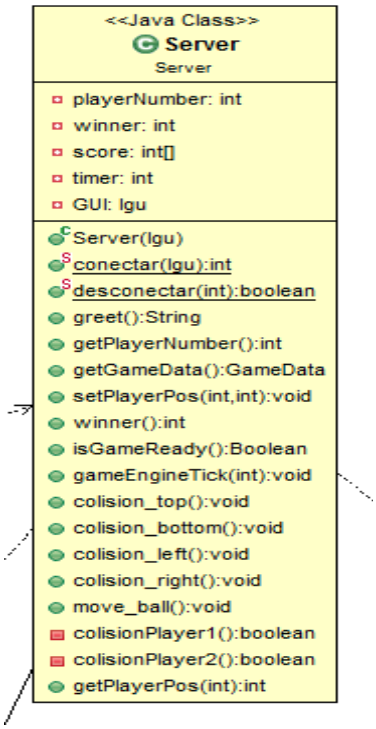
```java
@Override
public void keyTyped(KeyEvent e) {
    try {
        GameData gameData=Client.stub.getGameData();
        if(GS.getPlayer() == 1 || GS.getPlayer() == 2){
            if(e.getKeyChar()=='a'){
                if(gameData.getPos(GS.getPlayer())>0)
                    Client.stub.setPlayerPos(GS.getPlayer(), Client.stub.getPlayerPos(GS.getPlayer())-5);
            }
            if(e.getKeyChar()=='q'){
                if(gameData.getPos(GS.getPlayer())<500-GameData.racketHeigth)
                    Client.stub.setPlayerPos(GS.getPlayer(), Client.stub.getPlayerPos(GS.getPlayer())+5);
            }
            if(e.getKeyChar()=='i'){
                System.err.println("He apretado la i madafaka!");
                this.GS.setKey_status(new key_status_normal(this.GS));
            }
        }
    } catch (RemoteException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}
```

➔ Key_status_normal implements Game_Scene_key_status

```
@Override
public void keyTyped(KeyEvent e) {
    try {
        GameData gameData=Client.stub.getGameData();
        if(GS.getPlayer() == 1 || GS.getPlayer() == 2){
            if(e.getKeyChar()=='q'){
                if(gameData.getPos(GS.getPlayer())>0)
                    Client.stub.setPlayerPos(GS.getPlayer(), Client.stub.getPlayerPos(GS.getPlayer())-5);
            }
            if(e.getKeyChar()=='a'){
                if(gameData.getPos(GS.getPlayer())<500-GameData.racketHeigth)
                    Client.stub.setPlayerPos(GS.getPlayer(), Client.stub.getPlayerPos(GS.getPlayer())+5);
            }
            if(e.getKeyChar()=='i'){
                this.GS.setKey_status(new key_status_inverse(this.GS));
            }

        }
    } catch (RemoteException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
}
```

Patrón OBSERVADOR:

Junto con la refactorización aplicada para el cálculo del sistema de colisiones, hemos añadido el patrón observador en la Igu del servidor (Server.Igu) para poder mostrar el resultado cada vez que este cambie.




```
public void colision_left(){
    if(gameData.getPosBallx()<0){
        score=gameData.getScore();
        score[1]++;
        this.GUI.showGameStatus(this.gameData);
        gameData.setScore(score);
        gameData.resetGame();
    }
}
public void colision_right(){
    if(gameData.getPosBallx()>GameData.frameWidth){
        score=gameData.getScore();
        score[0]++;
        this.GUI.showGameStatus(this.gameData);
        gameData.setScore(score);
        gameData.resetGame();
    }

}
```

```
public void showGameStatus(Common.GameData gameData){
    textArea.setText(textArea.getText()+"---- IP ----\n");
    showIps();
    textArea.setText(textArea.getText()+"---- Juego: ----\n");
    textArea.setText(String.valueOf(gameData.getScore()[0])+" - "+ String.valueOf(gameData.getScore()[1]));

}
```

Patrón ENVOLTORIO O WRAPPER:

Las clases racket y ball son en sí, representaciones de la posición de la raqueta y la pelota en la pantalla. Es por ello, que se trata de una clase primaria PosEntity (que representa una posición en coordenadas X, Y). Es por ello, que utilizamos las clases Racket y Ball para que sea más comprensible su edición.

En el caso de la raqueta y la pelota, hemos cambiado la interfaz de uso: de PosEntity.{get/set}X a Racket.{get/set}PosX y de PosEntity.{get,set}Y a Racket.{get,set}PosY, para facilitar su compresión.

```java
public class Racket extends PosEntity implements Serializable{
    /**
     *
     */
    private static final long serialVersionUID = 1L;


    //Constructors
    public Racket(){
        super();
    }
    public Racket(int posx, int posy){
        super(posx, posy);
    }

    public int getPosX(){return this.getX();}
    public int getPosY(){return this.getY();}
    public void setPosX(int x){this.setX(x);}
    public void setPosY(int y){this.setY(y);}
}
```

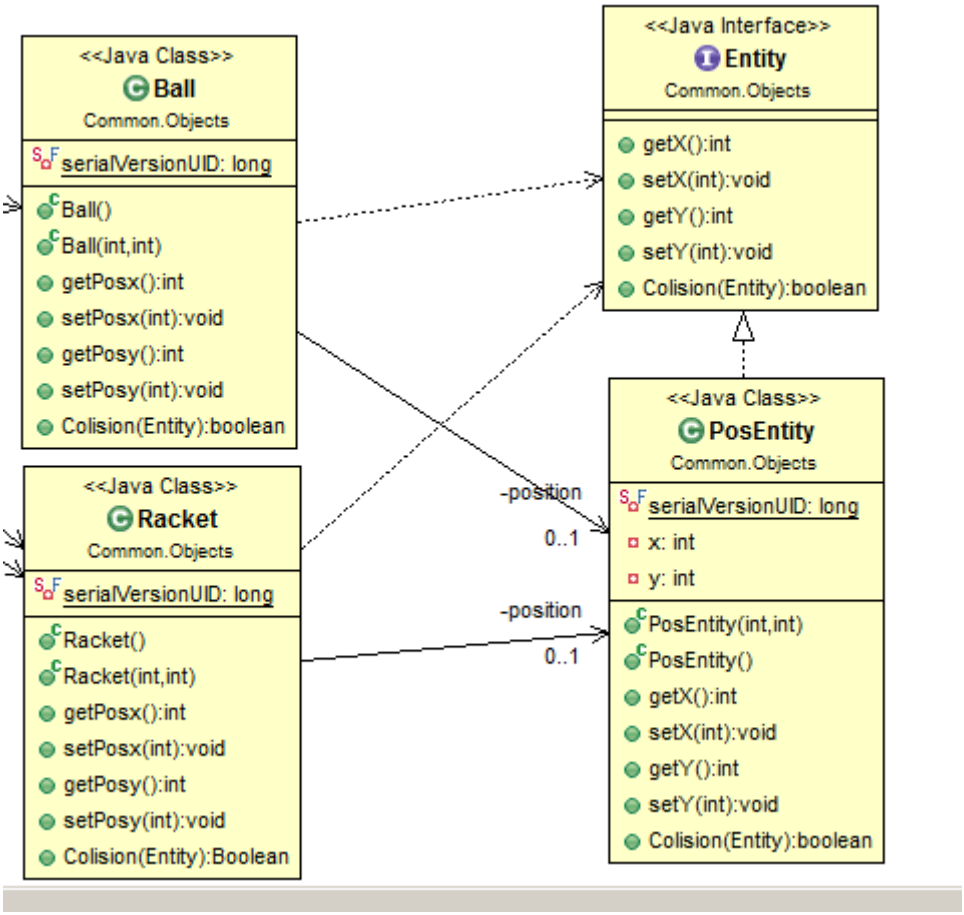
```java
public class PosEntity implements Entity{
    private static final long serialVersionUID = 1L;
    private int x,y;

    public PosEntity(int x, int y) {
        super();
        this.x = x;
        this.y = y;
    }
    public PosEntity() { new PosEntity(0,0); }

    @Override
    public int getX()       {       return x;   }
    @Override
    public void setX(int x) {       this.x = x; }

    @Override
    public int getY()       {       return y;   }

    @Override
    public void setY(int y) {       this.y = y; }

    @Override
    public boolean Colision (Entity entity){
        return (entity.getX() == this.getX()) || (entity.getY() == this.getY()) ;
    }
}
```

4. REFACTORIZACIÓN APLICADA

Al realizar el juego desde 0, no hemos tenido demasiados problemas a la hora de refactorizar código. Es por ello, que sólo ponemos 2 casos de refactorización del código.

Objetos RACKET Y BALL.

Ambos objetos tienen en común que “son” una posición en la pantalla que puede ir cambiando. Por ello, en vez de copiar ambas clases, hemos creado una clase “PosEntity” como entidad de posición.



Método demasiado largo:

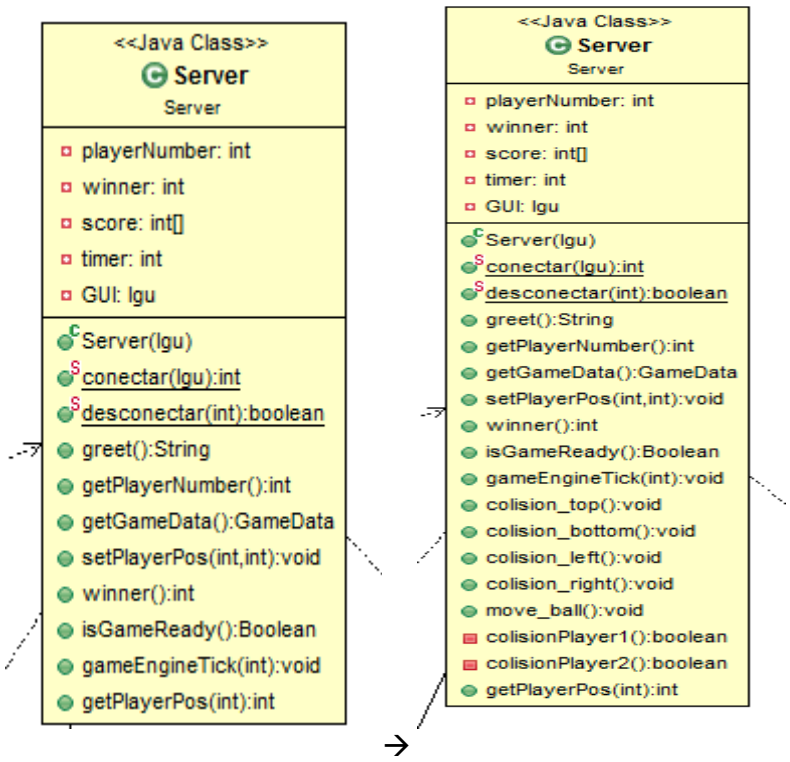
En el segundo caso, se trata del método gameEngineTick, de la clase Server. Este método es el encargado del sistema de colisiones de la pelota. Para contar todos los casos, se había hecho un método muy largo.

```
@Override
public void gameEngineTick(int player) throws RemoteException {
	if(player==1){
		score=gameData.getScore();
		//Colision techo
		if(gameData.getPosBally() < 0){
			gameData.setY_sig(true);
		}
		//Colision suelo
		if(gameData.getPosBally() > GameData.frameHeigth+
(GameData.ballSize)-102){
			gameData.setY_sig(false);
			//soundEffects[1]=true;
		}
		//Colisón derecha
		if(gameData.getPosBallx()>GameData.frameWidth){
			score[0]++;
			gameData.resetGame();
		}
		//Colisión izquierda
		if(gameData.getPosBallx()<0){
```

```
                        score[1]++;
                        gameData.resetGame();
                  }

                  //Colisón paleta jugador 2
                  if(colisionPlayer2()){
                        gameData.setX_sig(false);
                  }
                  //Colisón paleta jugador 1
                  if(colisionPlayer1()){
                        gameData.setX_sig(true);
                  }
                  //--------------Ball-Move-------------------------//
                  if(gameData.getX_sig() && gameData.getY_sig()){
      gameData.setPosBallx(gameData.getPosBallx()+gameData.getSpeed());
                        gameData.setPosBally(gameData.getPosBally()
+gameData.getSpeed());
                  }
                  if(!gameData.getX_sig() && gameData.getY_sig()){
                        gameData.setPosBallx(gameData.getPosBallx()-
gameData.getSpeed());
                        gameData.setPosBally(gameData.getPosBally()
+gameData.getSpeed());
                  }
                  if(gameData.getX_sig() && !gameData.getY_sig()){
                        gameData.setPosBallx(gameData.getPosBallx()
+gameData.getSpeed());
                        gameData.setPosBally(gameData.getPosBally()-
gameData.getSpeed());
                  }
                  if(!gameData.getX_sig() && !gameData.getY_sig()){
                        gameData.setPosBallx(gameData.getPosBallx()-
gameData.getSpeed());
                        gameData.setPosBally(gameData.getPosBally()-
gameData.getSpeed());
                  }

                  //-------------Score Engine-----------------//
                  if(score[0]>=5)
                        this.winner=1;
                  if(score[1]>=5)
                        this.winner=2;

                  gameData.setScore(score);
                  //------------------SpeedIncrease-------------//
                  if(timer>900){
                        gameData.setSpeed(gameData.getSpeed()+2);
                        timer=0;
                  }
                  timer++;
                  //=========================================================
====================================//
            }
      }

      private boolean colisionPlayer1() {
            if(gameData.getPosBallx() > 10 && gameData.getPosBallx() < 10 +
GameData.racketWidth){
```

```
                    if(gameData.getPosBally() <
gameData.getPos(1)+GameData.racketHeigth && gameData.getPosBally() >
gameData.getPos(1))
                            return true;
                    if(gameData.getPosBally()+GameData.ballSize <
gameData.getPos(1)+GameData.racketHeigth && gameData.getPosBally()
+GameData.ballSize > gameData.getPos(1))
                            return true;
             }
             return false;
      }

      Private boolean colisionPlayer2() {
             if(gameData.getPosBallx()+GameData.ballSize >
GameData.frameWidth-30 && gameData.getPosBallx()+GameData.ballSize <
GameData.frameWidth - 10){
                    if(gameData.getPosBally() <
gameData.getPos(2)+GameData.racketHeigth && gameData.getPosBally() >
gameData.getPos(2))
                            return true;
                    if(gameData.getPosBally()+GameData.ballSize <
gameData.getPos(2)+GameData.racketHeigth && gameData.getPosBally()
+GameData.ballSize > gameData.getPos(2))
                            return true;
             }
             return false;
      }
```

Tras el análisis de casos, dividimos el método en (tal y como ya aparece arriba) en varios, tanto para analizar la colisión con las paletas de los jugadores, como el análisis de colisiones en los marcos de la ventana, quedando finalmente:



5. PRUEBAS UNITARIAS

Puesto que hemos refactorizado las colisiones, nos parecía adecuado probar las colisiones con los marcos del frame de juego. Para ello, debemos comprobar:

1) Colisión en marco derecho →`test_colision_right`
   a) La pelota vuelve a la posición inicial.
   b) El jugador 1 (Score[0] ha conseguido un punto).

```
@Test
public void test_colision_right(){
    Igu i = new Igu("Junit_test");
    i.setVisible(false);

    Server S = new Server(i);
    GameData g = S.getGD();

    g.setPosBallx(10000);
    S.colision_right();

    int[] score = g.getScore();
    System.out.println(g.getPosBallx());
    assertTrue(score[0] == 1 &&  g.getPosBallx() == 390);
}
```

2) Colisión en marco Izquierdo:
   a. La pelota vuelve a su posición original.
   b. El jugador 2 (Score[1]) ha conseguido un punto.

```
@Test
public void test_colision_left(){
    Igu i = new Igu("Junit_test");
    i.setVisible(false);

    Server S = new Server(i);
    GameData g = S.getGD();

    assertFalse(g == null);

    g.setPosBallx(-3);
    S.colision_left();

    int[] score = g.getScore();
    assertTrue(score[1] == 1 && g.getPosBallx() == 390);
}
```

3) Colisión en marco superior:
    a. La pelota vuelve a posición (0)
    b. La dirección de la pelota es descendente

```
@Test
public void test_colision_top(){
    Igu i = new Igu("Junit_test");
    i.setVisible(false);

    Server S = new Server(i);
    GameData g = S.getGD();

    assertFalse(g == null);

    g.setPosBally(-3);
    S.colision_top();

    assertTrue(g.getPosBally() == 0 && g.getY_sig());
}
```

4) Colisión en marco inferior:
    a. La pelota vuelve a posición más baja posible
    b. La dirección de la pelota es ascendente

```
@Test
public void test_colision_bottom(){
    Igu i = new Igu("Junit_test");
    i.setVisible(false);

    Server S = new Server(i);
    GameData g = S.getGD();

    assertFalse(g == null);

    g.setPosBally(10000);
    S.colision_bottom();

    int max_y = GameData.frameHeigth+(GameData.ballSize)-102;

    System.err.println(String.valueOf(g.getPosBally() + "--"+String.valueOf(max_y)));
    assertTrue(g.getPosBally() == max_y  && !g.getY_sig());
}
```