# Lab report on my PingPong game app.

## *Preamble..*

*Our team never really got working. One member did suggest doing the Squash game and I started on the UML for that. However he changed his mind to Pong game and I followed that lead.*

*Unfortunately, while I wanted to use the code framework given us by Thomas and Ralf, the other contributor decided to use a AWL/Swing framework. We decided therefore that we needed to work separately*

*I only became aware of the game layout and the need to create a lab report when I was about to submit the zip file so this report has been create retrospectively. And the game itself plays against the computer not another player. It is now too late to make that change now! Sorry*

## The Journal

I used the TheProject framework supplied by Thomas and Ralf as the basis for the application. I examined and coded the entire example given us and I therefore also made use of some ideas in the Capstone1-counter3 and capstone1-counter5 examples.

In particular, I placed and moved the ball in the draw() method of the TheApp object. This was analogous to the small white rectangle placed randomly in capstone1-counter5 example.

The playing field/table is drawn left to right with the player paddle to the left and the computer paddle to the right rather than top to bottom.

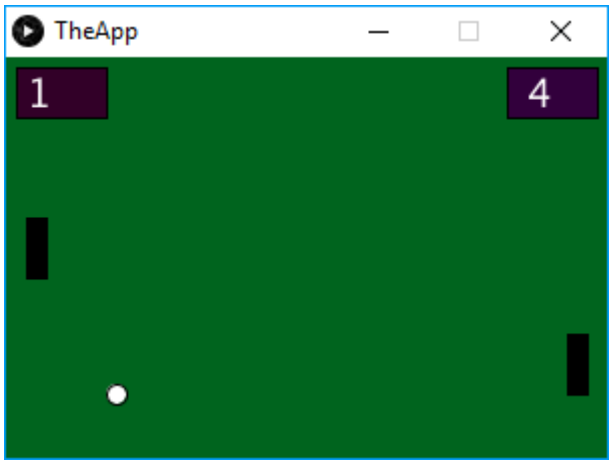



Fig.1 the final App.

capstone1-counter5 code

```
@Override
	public void draw() {  // draw() loops forever, until stopped
		background(204);
		fill(255);
		rect(random(100),random(100), 10, 10);
		interactiveCounter.update();
	}
```

And my code

```
public void draw() {  // draw() loops forever, until stopped
              background(0,100,50);
              fill(255);
              rect(ballX,ballY,10,10);
              incrementX();
              incrementY();

       }
```

I called two methods in the TheApp class to move the ball in the x and Y direction

```
              background(0,100,50);
              player.update();
              computer.update();
              fill(255);
              rect(ballX,ballY,10,10);
              incrementX();
              incrementY();
       }
```

I used subclasses of the Abstractview to draw and manage the Paddles/Bats, see below. As I hadn't yet created an InteractiveComponent I called their update methods directly.

The incrementX(0 and incrementY() methods move the ball around. Initially these were very similar i.e. they simply bounced the ball from the surrounf=ding walls.

```
private int incrementX() {
              ballX = ballX + deltaX;
              if(ballX < 0) {
                     ballX = ballX * -1;
                     deltaX = deltaX * -1;
              }
              if(ballX > tableWidth) {
                     ballX = (tableWidth*2) -ballX;
                     deltaX = deltaX * -1;
           }
              return ballX;
       }
```

ballX marks the X position of the ball and deltaX is the increment to move it by. If the ball hits an end wall then deltaX is negated to ensure the ball moves in the opposite direction.

Later these methods were changed so that in the  incrementX() method the ball would  bounce from the paddles and would also register a score if the ball moves to either end wall. And the incrementY() method was changed to call a move method to move the computer paddle automatically.

The final code is given here

```
private int incrementX() {

              // change the ball direction is it collides with either paddle
              if (interactivePingPong.hasCollision(ballX, ballY)) {
                     deltaX = deltaX * -1;
              }
```

```
            // check if ball hits left end wall
            if(ballX <= 0) {
                  // should update score for computer
                  interactivePingPong.handleComputerScoreEvent();
                  // ballX = ballX * -1;
                  deltaX = deltaX * -1;
                  // set ball in centre again
                  ballX = 150;
                  ballY = 100;
            }
            if(ballX >= tableWidth) {
                  // should update score for player
                  interactivePingPong.handleUserScoreEvent();
                  // ballX = (tableWidth*2) -ballX;
                  deltaX = deltaX * -1;
                  // set ball in centre again
                  ballX = 100;
                  ballY = 100;

          }
            // move in x direction
            ballX = ballX + deltaX;
            return ballX;
      }

private int incrementY() {
            // check if ball hits top wall
            if(ballY < 0) {
                  ballY = ballY * -1;
                  deltaY = deltaY * -1;
            }
            // or bottom wall
            if(ballY >= (tableHeight -10)) {
                  ballY = ((tableHeight-10) *2) -ballY;
                  deltaY = deltaY * -1;
          }
            // move computerbat if necessary
            interactivePingPong.moveComputerbat(ballY);
            // move computerbat as necessary.
            ballY = ballY + deltaY;
            return ballY;
      }
```

The views of the paddles were derived from AbstractView, via an AbstractPaddleView.. I also derived two other views from AbstractView, the ScoreView to display the scoreboard and the UserWinView to display a message when the user won ( or lost! .The name could be better).

I used the Counter class to maintain the score. Other values , such as the size of the table and the ball position could have been maintained here but in the end I left then in the TheAPP class. This could be tidied up though.

I created a Controller component and also an InteractiveComponent, InteractivePingPong.  I used the builder model , as shown in capstone1-counter5 t oconstruct the InteractivePingPong. I feel this is a very elegant solution and I

had never used this type of class creation structure before. Is is definitely easier to do rather than have a class with a contructor taking numerous parameter. Following capstone1-counter5 example I also used a static method in a construct class to build the component. Have never coded a static method this way before either so was good learning for me.

With the controller, which handled feed from the views, I had a problem. The Controller itself had no reference to all the views and so it passed on requests to the InteractivePingPong which then responded to each input. Eventually the Controller component became redundant , with the InteractivePingPong object handling interaction and updating the model, Counter object, as necessary. At this stage the Controller is still part of the project but I will refactor to remove it .

The InteractiveComponent, InteractivePingPong manages the game. It holds references to  each view , to the model, and the controller( althought not needed ) . Other than the methods needed by the Builder class to construct it, it also has the following methods.

```
	@Override
	public void handleLeftKeyevent()
		userBat.moveLeft(10,0);
	}
```

These is also a method to handle right key events

```
	@Override
	public void handleUserScoreEvent() {
		counter.updateUserScore();
		if (counter.isGameOver()) {
			scoreboard.update();
			win.update();
		}

	}
```

These is also an equivalent method to handle ComputerScoreEvents

The following method calls each paddles methods to see if a collision has happened.

```
	@Override
	public boolean hasCollision(int x,int y) {
		boolean collision = false;
		if (userBat.hasCollided(x, y) || computerBat.hasCollided(x, y)) {
			collision = true;
		}
		return collision;
	}
```

The following method moves the computerBat automatically. However it only moves it one pixel at a time and then only of the paddle is more than 20 pixels from the axis of the ball. Otherwise the computer paddle is too fast and accurate and the computer would always win.

```
@Override
	public void moveComputerbat(int y) {
```

```
            int batcentreY = computerBat.Ypos +15;
            int ballcentreY = y +5;

            if(batcentreY != ballcentreY) {
                  if ((batcentreY- ballcentreY) > 20 ) {
                  computerBat.moveLeft(1,0);
                  } else {
                  if (ballcentreY - batcentreY > 20) {
                        computerBat.moveRight(1,200);}
                  }
            }

      }
```

This class  also contains these methods which are handled by the controller. However these are not needed and I will refactor to remove then and then remove the Controller class.

```
      @Override
      public void handleEvent()

      @Override
      public void handleMouseEvent()

      @Override
      public void handleKeyEvent()
```

The PaddleView class is responsible for the movement of the paddle. This class draws the paddle at the position required in its update method.

The class has methods to move the paddles left or right within the range of the playing field (note that the playing field in my app is from left to right and therefore the paddle move from the top to the bottom of the screen)

```
public void moveLeft(int delta, int minimum) {
            Ypos = Math.max(minimum, (Ypos -delta));
      }
```

Delta is the amount to move the paddle by and minimum os the limit of the sceen , which is usually 0 but could by a bit more if for instance  the playing area had a frame.

```
public void moveRight(int delta, int maximum) {
            Ypos = Math.min((maximum-30), (Ypos+delta));
      }
```

In this case the maximum is the size of the screen . Passing minimum ansd maximum allow for the method to handle and size screen and ensure the paddle does not exceed the playing area.

And also a method to detect collisions.

```
public boolean hasCollided(int x, int y) {
            boolean collision = false;
            // get centre of ball
```

```
            int ballcentreX = x+5;
            int ballcentreY = y+5;
            // get centre of paddle
            int paddlecentreX = Xpos+5;
            int paddlecentreY = Ypos+15;
            int xdiff = Math.abs(ballcentreX - paddlecentreX);
            int ydiff = Math.abs(ballcentreY - paddlecentreY);
            // checks a box a couple of pixels around the bat
            if((xdiff <=10)&&(ydiff <=15)) {
                  collision = true;
            }
            return collision;
      }
```

The methis is passed the x and y cooridinates of the ball. Xpos and Ypos are the position of the paddle. If the distance between them is less than the given values then they have collided.

## Future development

Ideally, the game should have an introduction and the user shouls be allowed set the no of balls to play. At present this is set at 10. It should be possible to allow two players to play and also to allow the player to choose which players will play.

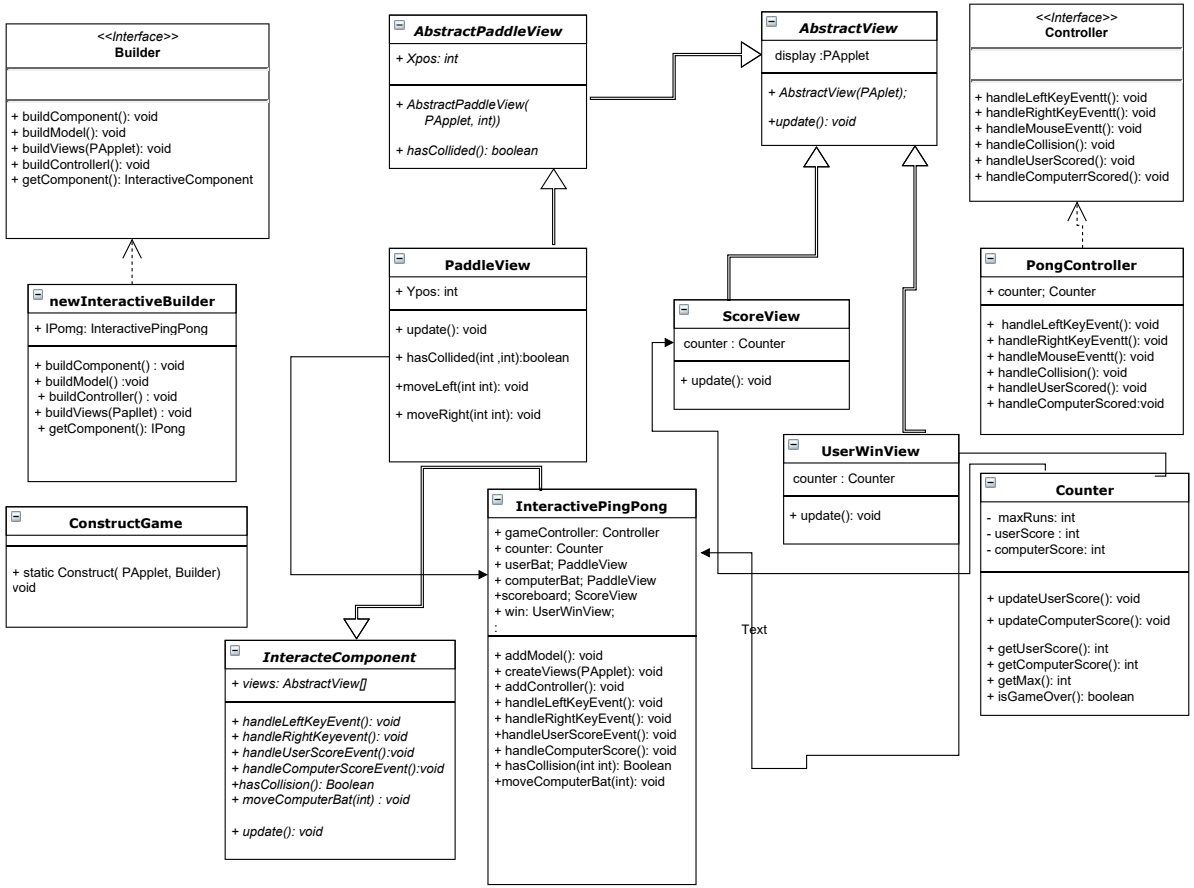
I should move all data into the Counter class, . All interactions are handled in the InteractivePingPong class and I could remove the Comtroller class completely.

Problems

I have had problems using Github and Draw.io , all down to my own inexperience. I would have liked a little more time to become more familiar with these tools

Finally a modified UML.

This was done after the event . As I explained in the Preamble above, I initially submitted a UML for the Squash game.


<<Interface>>
Builder
+ buildComponent(): void
+ buildModel(): void
+ buildViews(PApplet): void
+ buildControllerl(): void
+ getComponent(): InteractiveComponent
AbstractPaddleView
+ Xpos: int
+ AbstractPaddleView( PApplet, int))
+ hasCollided(): boolean
AbstractView
display :PApplet
+ AbstractView(PAplet);
+update(): void
<<Interface>>
Controller
+ handleLeftKeyEventt(): void
+ handleRightKeyEventt(): void
+ handleMouseEventt(): void
+ handleCollision(): void
+ handleUserScored(): void
+ handleComputerrScored(): void
newInteractiveBuilder
+ IPomg: InteractivePingPong
+ buildComponent() : void
+ buildModel() :void
+ buildController() : void
+ buildViews(Papllet) : void
+ getComponent(): IPong
PaddleView
+ Ypos: int
+ update(): void
+ hasCollided(int ,int):boolean
+moveLeft(int int): void
+ moveRight(int int): void
ScoreView
counter : Counter
+ update(): void
PongController
+ counter; Counter
+ handleLeftKeyEvent(): void
+ handleRightKeyEventt(): void
+ handleMouseEventt(): void
+ handleCollision(): void
+ handleUserScored(): void
+ handleComputerScored:void
ConstructGame
+ static Construct( PApplet, Builder) void
InteractivePingPong
+ gameController: Controller
+ counter: Counter
+ userBat; PaddleView
+ computerBat; PaddleView
+scoreboard; ScoreView
+ win: UserWinView;
:
+ addModel(): void
+ createViews(PApplet): void
+ addController(): void
+ handleLeftKeyEvent(): void
+ handleRightKeyEvent(): void
+handleUserScoreEvent(): void
+ handleComputerScore(): void
+ hasCollision(int int): Boolean
+moveComputerBat(int): void
UserWinView
counter : Counter
+ update(): void
Counter
- maxRuns: int
- userScore : int
- computerScore: int
+ updateUserScore(): void
+ updateComputerScore(): void
+ getUserScore(): int
+ getComputerScore(): int
+ getMax(): int
+ isGameOver(): boolean
InteracteComponent
+ views: AbstractView[]
+ handleLeftKeyEvent(): void
+ handleRightKeyevent(): void
+ handleUserScoreEvent():void
+ handleComputerScoreEvent():void
+hasCollision(): Boolean
+ moveComputerBat(int) : void
+ update(): void
Text