**SW Devt IV – Advanced .NET 420-411-DW  
Lab Exercise 2 - Single-player Pong part 1**

We are going to write a small game, taking the same approach that we will be taking with the project: write business classes, unit test the business classes, then add the UI layer. We won’t actually unit test the UI: testing GUI is a pain in the neck ☹

If you are working at home or on a laptop, have either Windows 7, Windows 8.1 or Windows 10, and want MonoGame to work with your Visual Studio 2015 please do this:

- make sure you have VS 2015 with Update 3 (Windows 10 SDK is included)

- install update (Win 7 and 8.1 only): <https://support.microsoft.com/en-ca/kb/2999226>

- install Monogame framework <http://www.monogame.net/downloads/> choose MonoGame 3.5 for VisualStudio. You don’t need to install the templates for previous versions of Visual Studio.

- If you have a Windows 7 or 8.1 system: apply updates to DirectX <https://www.microsoft.com/en-us/download/confirmation.aspx?id=35>

**Game specification**: We are building a single paddle version of Pong: a paddle on the bottom is controlled by the user, and has to be placed under a falling ball so the ball doesn’t touch the “ground”. If the ball is bounced off the paddle, it will deflect, and bounce off the sides and the top. If it touches the ground – game over!!

**Part 1**

We are building a **Windows– Class Library** to hold the Pong business classes - call it PongLibrary, but the solution Pong. These classes are GUI-independent, and contain only the Pong business logic. We want to be able to access the useful Monogame framework classes, so we need to make sure that the project references Monogame:

In the Solutions Explorer > Expand the PongLibrary project > right click on References. Add a reference > Assemblies > Extensions on the left, and select a MonoGame.Framework (there are multiple, hold your mouse over and select the one in Windows/MonoGame.Framework.dll. Make sure you check the checkbox!).

You can now access XNA classes (XNA was the gaming framework offered by Microsoft that is available as open-source by Monogame. Most XNA documentation from Microsoft is valid). Add this directive to the start of your classes to access XNA classes without full qualification:

using Microsoft.Xna.Framework;

**Design**: We want to separate the business logic from the GUI, input and game loop logic. We will refine our design through the next few labs. To start, we need business-layer classes. Note: this is probably overkill for a simple application like Pong, but good design becomes a good habit when you persist ☺ The two business layer classes:

* + Ball class: encapsulates the behavior and state of a Ball
  + Paddle class: encapsulates the behavior and state of a Paddle

In this lab we will start to build two business classes in a library. We'll revisit these classes when we implement the Observer design pattern.

Let’s tackle the Paddle class design first. The private fields are indicated below, but this is really an implementation decision, so feel free to change.

**Paddle**

- speed: int <<readonly>>

- screenWidth : int

- boundingBox: Rectangle

<<constructors>>

+ Paddle(paddleWidth: int, paddleHeight: int, screenWidth : int, screenHeight : int , speed: int)

<<properties>>

+ BoundingBox : Rectangle <<public get only>>

<<custom methods>>

+ MoveLeft() : void

+ MoveRight() : void

The [Microsoft.Xna.Framework.Rectangle](http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.rectangle.aspx) structure defines the height and width of an object, as well as the X and Y coordinates of the top-left corner (as int). We will use it to hold all positional information of the Paddle as well as for simple discrete collision detection.

The fields:

* speed is a readonly int that is used to determine how fast (i.e., how many pixels) the Paddle moves every time the MoveLeft or MoveRight methods are called. (remember C# naming conventions do not using screaming caps for readonly and const.
* the screenWidth gives the Paddle the dimension of the playing screen. It is used to calculate if the paddle is allowed to move that far right (it is blocked from going off screen)
* the boundingBox Rectangle struct is instantiated with the paddle's height and width, as well as top left position.

The constructor instantiates the boundingBox’s x and y such that the Paddle is centered at the bottom of the screen (and visible!). Remember, the origin is the top left corner.

(0,0)

screenHeight

screenWidth

x,y

paddleHeight

paddleWidth

Finally the MoveLeft and MoveRight methods change the x by the amount of the speed, but making sure that the Paddle does not go out of bounds. The [Microsoft.Xna.Framework.MathHelper](http://msdn.microsoft.com/en-us/library/bb197889.aspx) class includes many static members, including Max, Min and Clamp methods (clamp restricts a value to a min-max range).

The Ball should be circular, however, we are going to consider it to be a square/rectangle for simple collision detection. As such, it has many similar fields/attributes as the Paddle, but it needs to know its screenHeight also as well as its velocity. You can put the screen dimensions in a Rectangle struct to help with collision detection with the edges. Unlike the Paddle’s speed, the velocity of the ball needs to be changeable and track speed in both the x and y directions; a Vector2 struct may be useful. The constructor will initialize the Ball’s position to be the top center of the screen.

The custom methods are different between the Paddle and the Ball: the Paddle can be moved left or right, while the Ball will simply Move. Its movement is dependent on its position and its velocity: add the velocity.X and velocity.Y coordinates to its x and y to move the ball along. But remember: you need to make sure the Ball doesn’t go off the screen! If the Ball is moving to the right, left or top, you need to let it get to the edge, then calculate how much was left of the velocity, and negate it to get the bounce – this is different from the Paddle, where you just stopped it at the end of the screen. Remember to account for the ball’s height and width for your code to work correctly. After bouncing, negate the velocity of the dimension (x or y) which bounced, so it continues to move in the opposite direction.

If the Ball hits the bottom, for now, set the velocity to (0,0) to basically end the game. We’ll come back to this.

Collision detection: decide whether it is the paddle’s or ball’s responsibility to detect collisions. If it is in the Ball, then each time it moves, it also has to check for collisions with the paddle. If the two Rectangles intersect with each other, it means the ball hit the paddle, so it needs to bounce.

For example:

if (this.BoundingBox.Intersects(paddle.BoundingBox))

this.Bounce();

Compile your code (i.e., build it).

**Part 2 – Unit testing**

Select the Solution in Solution Explorer, right-click - Add then choose new Project - Test - **Unit Test Project**. This uses the Microsoft unit test framework. Call it PongTests.

In the PongTests project, add a reference to the PongLibrary solution so you can access the library that you built: in Solution Explorer, select References in the PongTests project and then choose Add Reference... Projects>Solution> and check PongLibrary. Now add a reference to MonoGame.Framework (under Assembies-Extensions, check the correct one) so that you can see the classes/structs defined in Monogame and XNA (e.g., the Rectangle struct).

Rename UnitTest1.cs to PaddleTest.cs. You will create different test classes for each class that needs to be unit tested.

Examine each public method in the Paddle class, and consider how the behaviour can be tested, and how you know if the test case passed or not. If we had performed Test Driven Development, we would have written these test cases before the actual methods, since test cases completely document the expected behaviour.

For example, there are three scenarios with the MoveLeft method: completely move left by the amount of your speed, partially move left and stop at when the paddle's left hits 0, or not move at all if the paddle was already at 0.

This means we will write three methods in our unit test class for the method: MoveLeft\_EnoughSpace(), MoveLeft\_PartialSpace(), MoveLeft\_NoSpace().

Each method has three sections:

1. Arrange: perform all the set-up required (e.g., instantiate a new Paddle object) so that you can provoke the desired situation. You may have to call multiple methods/instantiate multiple objects before you are set-up.
2. Act: run the test; i.e., call the method that you are testing
3. Assert: check if the end result is as expected. This is typically done by calling the static methods of the Assert class or the StringAssert class

For example, the MoveLeft\_EnoughSpace() method may look like this:

[TestMethod]

public void MoveLeft\_EnoughSpace()

{

//arrange

int paddleW = 10;

int screenW = 20;

int speed = 4; //should end up at 1

Paddle paddle = new Paddle(paddleW, 0, screenW, 0, speed);

//act

paddle.MoveLeft();

int x = paddle.BoundingBox.X;

//assert

Assert.AreEqual( 1, x); //expected, actual

}

As you write your test cases you may notice a lot of repetitive code: follow the best practices and put these in private methods. A factory method for instantiation under different circumstances might also be a good idea.

Note: When you have more complicated test cases, where one object depends on another object, you want to make sure that you can unit test both objects separately and independently. Here’s where interfaces can be very useful. For example, testing the Ball object requires a Paddle object so that it can check for collisions; but actually what it really needs is something that implements IPaddle that provides a BoundingBox property. You can now define your own TestPaddle class that implements IPaddle and use the TestPaddle when arranging your Ball’s test cases. Programming to interfaces makes unit testing easier!

When you are ready to run the test cases - build the solution. If Test Explorer does not appear after a successful build, choose Test on the menu, then Windows, then Test Explorer. You should see the UnitTestExplorer window appears with your test cases listed in the Not Run Tests group. Choose Run All to run the test. As the test is running the status bar at the top of the window is animated. At the end of the test run, the bar turns green if all the test methods pass, or red if any of the tests fail.

If a test does fail, the test method is moved to the Failed Tests group. Select the method in Test Explorer to view the details at the bottom of the window.

The test result contains a message that describes the failure. For the AreEquals method, message displays what was expected (the (Expected<XXX>parameter) and what was actually received (the Actual<YYY> parameter).

Have fun!