Test Driven Development – Part I: TFC

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

In this first of the three part series on Test Driven Development, we focus on using NUnit to write our test cases and will illustrate the benefit of writing the test first, that is before writing the code. In Part II we will look at Mock objects and in Part III we will look at continuous integration.

A Problem Statement

Let's consider the following problem statement. Please do not read further. Spend about 10 minutes on the following assignment:

Draw a UML diagram or simply list the classes you would want to write for the following application.

We want to implement the Tick-Tack-Toe in this exercise. There are two users to the system. One will place an 'x' peg and the other an 'o' peg in cells. There are three rows and three columns. First a user must indicate whether the first player will use the 'x' peg or the 'o' peg. Then the first player is asked to place a peg on a cell. The player can only place on an empty cell. The game continues until a player has placed three pegs in a row, column or diagonally or there are no more empty cells left. If the game is won, the victory is announced. The application will keep track of the number of wins by each player. At any time, a user may request to view the statistics of the name of players and number of games each one has won.

Have you spent the 10 minutes thinking about the design of the system? OK, now you can read on.

Initial Thoughts on Classes

I have had the opportunity to use this exercise in my classes and at the symposiums where I speak. This article is based on those events and they all are more or less consistent. When discussing this, we generally came up with classes like Peg, Board, Cell, User, Player, Score, Statistics, and Rules to mention a few.

Years ago when I used to develop applications for the middle tier, I would sit down and write some test code or a UI to test the code I wrote. It was some what painstaking to test some of the code and I managed to do some marginal test until I gained confidence that the code is doing what it is supposed to do. Also, we sat down and came up with a design, drew some diagrams using notations (like UML or those that it derived its roots from). Once I put the code out for some one to integrate with, I would proceed with the next task. Days or even weeks later, I would hear from the surprised programmer integrating with my code as to why it seems to not work the way it was expected. At that point, it usually was more expensive to figure out what was going on and fix it.

Unit Testing is an Act of Design than Verification

To say that I have fallen in love with Unit Testing is an understatement. It has worked so well for my projects and I can't imagine developing applications without it. If you have not had a chance, I strongly recommend you to read the great books[1, 2, 3, 4] mentioned in the reference at the end of the article.

In Test First Coding, as we write the test code before writing the class, we are motivated to think about how our class will be used. Without it we focus more on implementation. This approach, on the other hand, let's us focus on how our class will be used. This leads to a design that is simpler and pragmatic. We can start out with a general understanding of the design with a high level UML diagram. However, once we get into developing the code, the unit tests can pretty much drive our design.

Of course, it is important that we use the Object-Oriented Design Principles[3] when writing the code. Without the principle, it would become a mere hack in my opinion.

How to write the test?

First think of what you want to test. There are at least three things we need to write the test for: positive, negative and exception. When thinking of a task, think of the positive, i.e., what it should do correctly assuming every thing is ideal. Then think of the negative, i.e., what could go wrong and how should the code behave. The exception is to think about possibility of alternate sequence of events that could happen and how the code should behave to accommodate those. The success of a positive test is when the code does what is expected. The success of negative and exception test may be if the code fails as expected.

Where to write a test?

Since we may be interested in testing not just the public methods but the internal methods as well, the test should be within the same project in .NET (in Java, within the same package). Of course, what if we want to test the private implementation of a class? Sure, we can write a test as a nested class in this case!

Task List

First we start out by writing a test list. This list will have one or more tests in it. Then we go through the list and pick the one that we can implement right away. As we start writing the test and the code, our mind (being a beautiful one) will think of other tests that we need to do. Do not write those tests when they come to your mind. Instead, put them at the end of the task list. It is important to continue working with the task on hand, but to jot down those thoughts that come to mind. Then you can go back and give due attention to those and take care of implementing those (if necessary).

Can we please start coding?

Enough said already. OK, OK, we can start coding. But first, let's create our task list. What do we want to test first? How about creating a board? That sounds good. But, what do we do after creating the board? Well, we should always write our test with assert in mind. We want to assert to see if the board was created fine. OK, we can do that, but that is trivial in most cases (unless we get a rather unlikely OutOfMemoryException). Humm,

what can we assert then? How about asserting the game is not over when we create the board. Alright, let's do that.

Task List

1. Create board

We first create a Blank Solution named TickTackToeApp. In it we create a C# Class Library project (You may have created a VB.NET project if you desired) named TickTackToeLib and in it create a class called TickTackToeTest with one method as shown below:

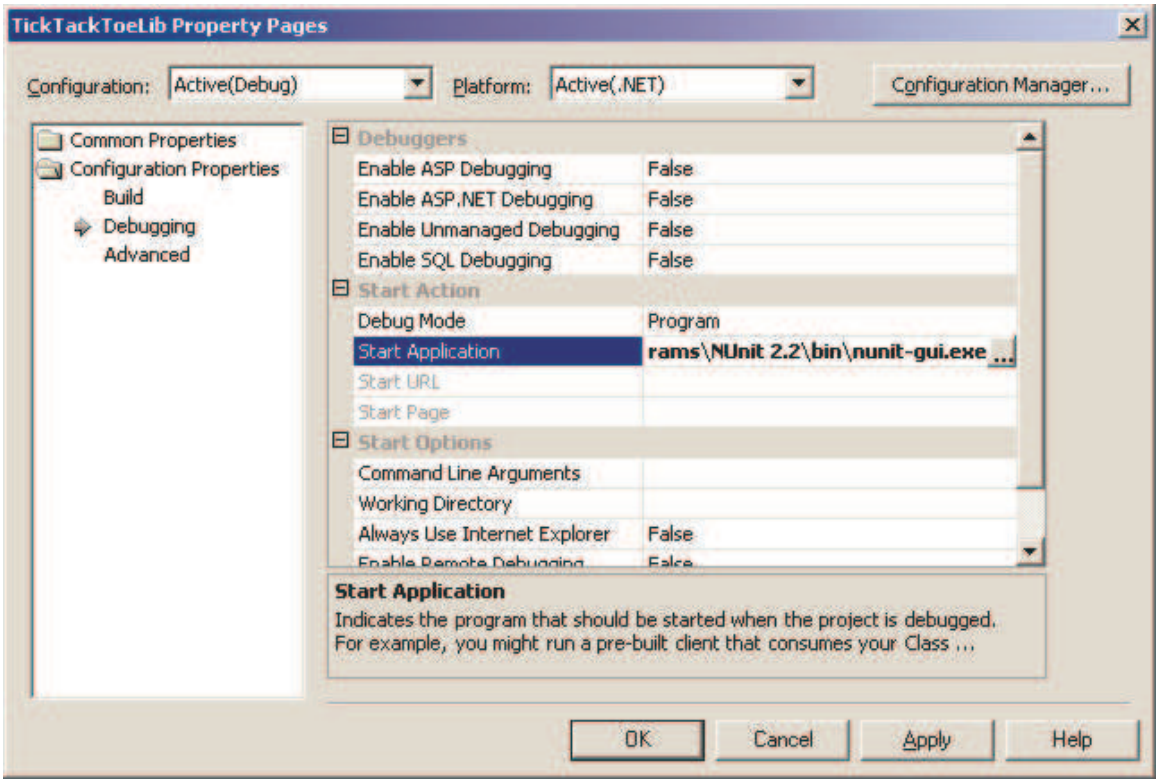
```
using System;
using NUnit.Framework;

namespace TickTackToeLib
{
	[TestFixture]
	public class TickTackToeTest
	{
		[Test]
		public void testCreateBoard()
		{
			TickTackToeBoard board = new TickTackToeBoard();
			Assert.IsNotNull(board);
			Assert.IsFalse(board.GameOver);
		}
	}
}
```

The TestFixture attribute tells us that the class is a Test case. The Test attribute tells us that the method is a test method. These are part of the NUnit framework. We have downloaded and installed NUnit 2.2[5]. We added a reference to nunit.framework.dll from the Global Assembly Cache (GAC). We now get a compilation error that the class TickTackToeBoard is not found. That is good. Our test case failed in a sense. Now we can create that class and implement the property GameOver as shown below:

```
using System;

namespace TickTackToeLib
{
	public class TickTackToeBoard
	{
		public bool GameOver
		{
			get { return false; }
		}

	}
}
```

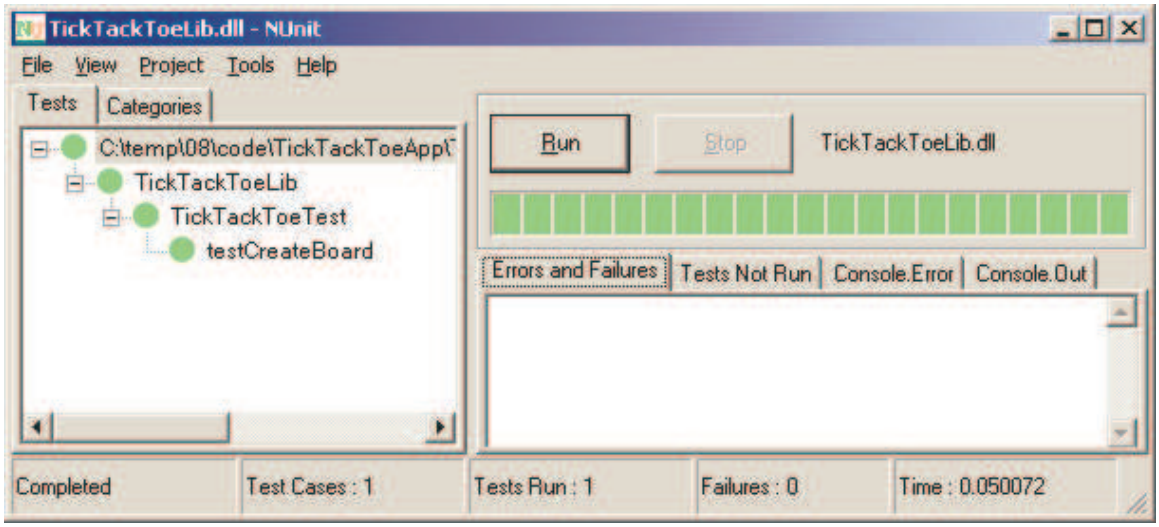
You look at the GameOver property and say, "Hum, what is the point of returning a false." Well, in TFC, we will lie our way through as much as we can. We provide only

implementation that is absolutely necessary. We will soon find that we can lie our way only for so long. OK, let's compile the code and make sure we have no more compilation errors. It is time to run our first test case. Go to solutions explorer and bring up the properties on the project. In the Debugging section, change the Debug Mode from Project to Program and click Apply. Then select the NUnitGui.exe in the Start Application as shown below:



Click OK and start the program (by hitting Ctrl + F5). This will bring the NUnitGui. You do not have to close this if you make change to the code. You can continue to edit and compile the code in studio and NUnitGui will automatically update the assembly and let you continue with your tests. When you start, NUnitGui will load up an assembly that it had loaded on a previous run (if any). You can click on File menu and click on Open menu item and open the TickTackToeLib.dll that you created.

Here is what we get when we run the program and click on Run.



There was one test case and it succeeded as indicated by the green bar. We successfully executed our first test. Let's proceed further. A look at our task list so far:

Task List

~~1. Create board~~

Now that we have the test succeed, what do we want to do next? Can we think of more tests to write? What come next to mind are the following tests:

Task List

~~1. Create board~~

2. Set First player
3. Set First player again
4. Set First Player after game starts

We not only want to test for setting the first player, we want to also think about how the code should behave if we set it again and if we set it after the game starts, that is after a peg has been placed.

Now, let's write the test for task #2. How should we write it? One participant suggested that we write:

Participant: board.SetFirstPlayer("Venkat");
Venkat: "Well, does the game care about the player being Venkat."
Participant: "Yeah!"
Venkat: "Why?"
Participant : "You sure want to know who the players are, don't you?"
Venkat: "I don't know at this point. I may need that later on, or may be not. Think of the YAGNI Principle. It stands for You Aren't Going to Need It (coined by Ronald E Jeffries). Do not build some thing that you are not sure you need at that moment."
Participant: "Hum…?"

(You will see the YAGNI at work in Part II for this feature) So, what should we do? Well, we may try

board.SetFirstPlayerPeg("X");

This will let the board (game) know that the first peg to be placed will be a "X" peg. That sounds good. But on a second thought, what if some one calls SetFirstPlayerPeg("Z"); Well Z is not a valid peg, so we need to write another test case to check for that. OK, let's go to the task list and add a test. Oh, wait a minute, there is another problem. If we allow them to set X or O, what if later on they want to use some other character? When we get it out, we need to check if we got a "X" or "O" as well. Is that really needed? What is we write some thing like:

board.IsFirstPlayerPegX(true);

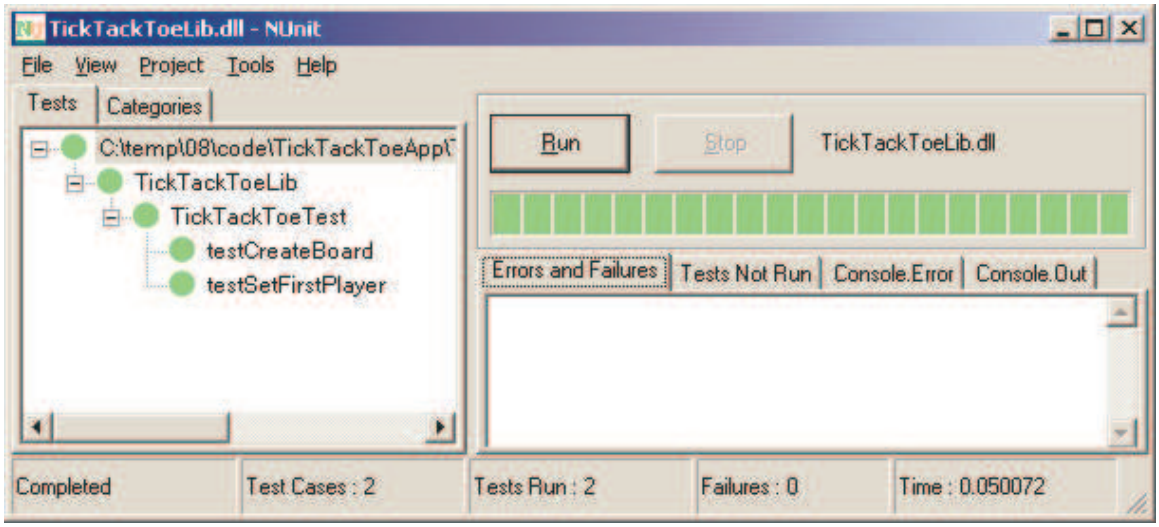
Then, we do not have to worry about that test and the characters used. "X" is more symbolic. This is a pretty simplified interface to the board isn't it? I like that, let's go for that, except we may use a property instead of a method. The test is shown below:

```
[Test]
public void testSetFirstPlayer()
{
      TickTackToeBoard board = new TickTackToeBoard();
      board.FirstPlayerPegIsX = true;
      Assert.IsTrue(board.FirstPlayerPegIsX);
}
```

We first set the first player peg to be X. Now, remember we have to write the test with assert in mind. So, we want to check if the peg to be placed is an X. Of course, this is arguable. Are we testing the set of the first player or are we testing the get of it? Well, if we really want to only test the set, then we may write a test in a nested class and set this property and then test some private member of the class. For now, I am going to accept the above as OK. We need to implement the property:

```
public bool FirstPlayerPegIsX
{
      get { return true; }
      set { }
}
```

That was quite a simple implementation. The get returns a true and the set does nothing. Let's compile and switch to NUnit and click on the Run button. We get:



Well, both the tests succeeded. Let's revisit our task list now and look at what we can pick from it:

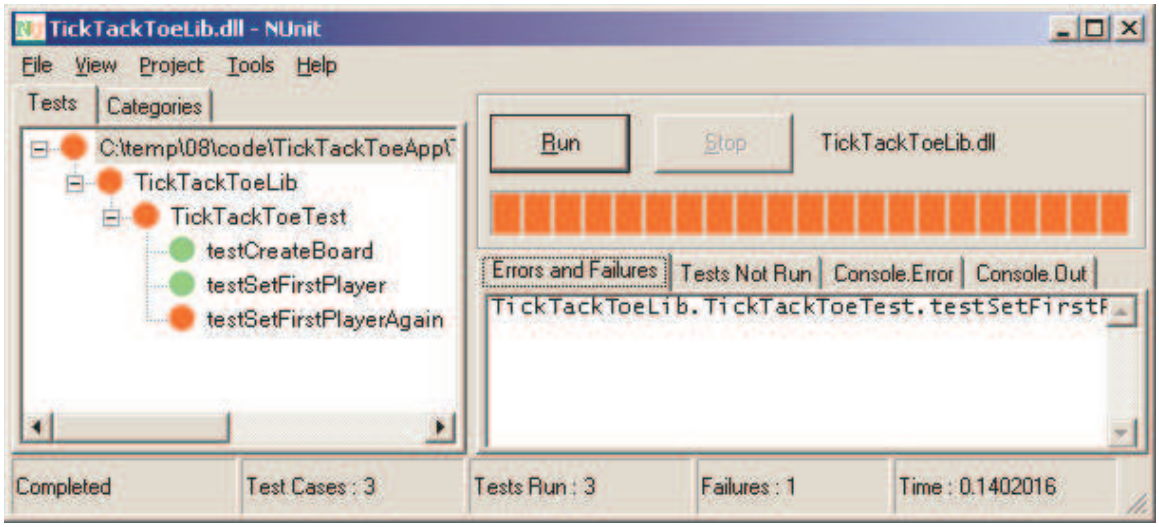
Task List

1. ~~Create board~~
2. ~~Set First player~~
3. Set First player again
4. Set First Player after game starts

We can't possibly pick task #4 as we have no idea how to do that yet. What about task #3. That looks like doable. So, here is the test for that:

```
[Test]
public void testSetFirstPlayerAgain()
{
      TickTackToeBoard board = new TickTackToeBoard();
      board.FirstPlayerPegIsX = true;
      Assert.IsTrue(board.FirstPlayerPegIsX);
      board.FirstPlayerPegIsX = false;
      Assert.IsFalse(board.FirstPlayerPegIsX);
}
```

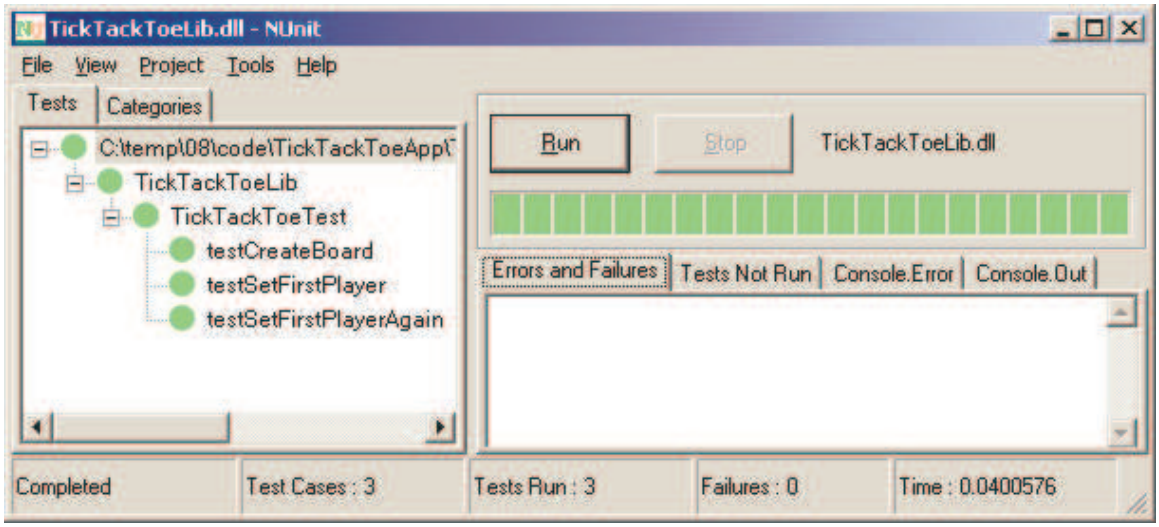
Now, we compile and since there was no compilation error, we switch to NUnit and click on Run. We get:



Well the third test failed. Why? Because, we expected a false, but the `FirstPlayerPegIsX` returned a true. Looking at the code we realize we could only lie our way through it for so long. Let's implement the property correctly now:

```
public class TickTackToeBoard
{
      private bool NextPlayerIsX;

      public bool GameOver
      {
            get { return false; }
      }

      public bool FirstPlayerPegIsX
      {
            get { return NextPlayerIsX; }
            set { NextPlayerIsX = value; }
      }

}
```

We introduced a private Boolean field named NextPlayerIsX and we are setting it to true within the FirstPlayerIsX property if the given input is true. Otherwise, set it to false. Now running the test case results in all the three tests succeeding as shown here:



Now is a good time to look at the code and refactor. Looking at the test class, we see that we are, at the beginning of each test case, creating an object of the Board repeatedly. That is a violation of the DRY[6] principle which stands for Don't Repeat Yourself. Well, we can move the object into the class as a member. The problem with that is if one test messes up the object, the tests following that may be messed up as well. We do not want that. We want tests to be isolated from one another. So, we want to create the object within each test. But, what about the DRY principle, should we just forget it? This is where the [SetUp] attribute comes in. A method that is declared with that attribute is executed at the beginning of each test. Similarly, a method marked with a [TearDown] is executed after each test executes. The test code is modified as shown below:

```
[TestFixture]
public class TickTackToeTest
{
      private TickTackToeBoard board;

      [SetUp]
      public void createBoard()
      {
            board = new TickTackToeBoard();
      }

      [Test]
      public void testCreateBoard()
      {
            Assert.IsNotNull(board);
            Assert.IsFalse(board.GameOver);
      }

      [Test]
      public void testSetFirstPlayer()
      {
            board.FirstPlayerPegIsX = true;
            Assert.IsTrue(board.FirstPlayerPegIsX);
      }
```

…

Let's look at the task list again:

Task List

1. ~~Create board~~
2. ~~Set First player~~
3. ~~Set First player again~~
4. Set First Player after game starts

The only test left on the list right now is the one I have no idea how to test yet. So, let's leave it there and get back to it later. We will write more tests now:

Task List

1. ~~Create board~~
2. ~~Set First player~~
3. ~~Set First player again~~
4. Set First Player after game starts
5. Place first peg
6. Place peg at occupied position
7. Place peg out of column range
8. Place peg out of row range

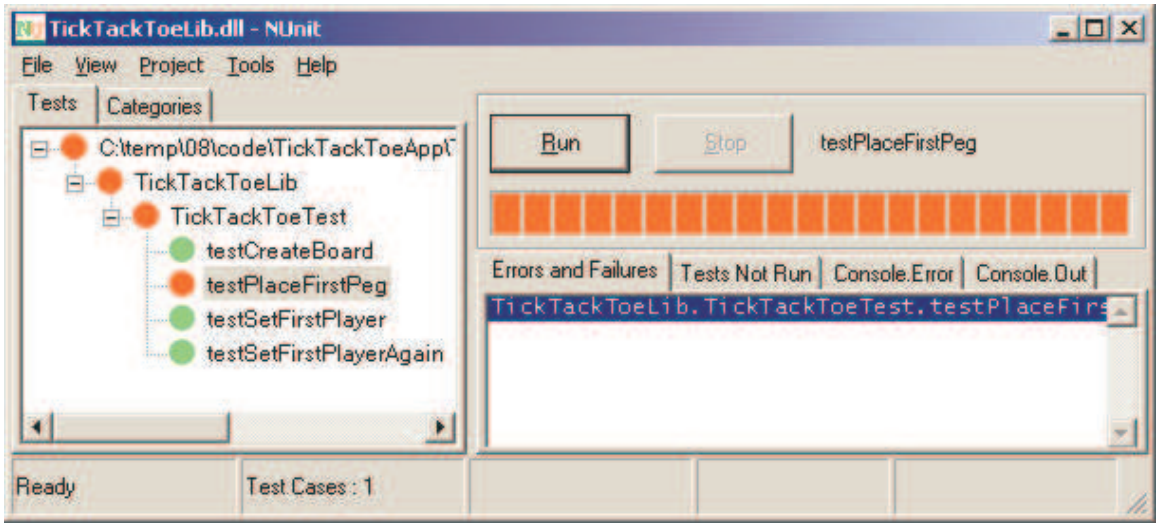
Typically we would write only one test at a time. We will go from writing test, implementing code, get a red bar on NUnit, get a green bar on NUnit, and refactor. Which test do we want to write now? Well, the obvious choice is task #5. So, here is the test:

```
[Test]
public void testPlaceFirstPeg()
{
	board.FirstPlayerPegIsX = true;
	board.PlacePeg(0, 1);
	Assert.IsTrue(board.PegAtPositionIsX(0, 1));
}
```

The first implementation of these methods is given below:

```
public void PlacePeg(int row, int column)
{
}

public bool PegAtPositionIsX(int row, int column)
{
	return false;
}
```

This will result in the test failing as shown here:



We now go from Red to Green with the following code:

```
private string[,] pegs
      = new string[,] {{"", "", ""}, {"", "", ""}, {"", "", ""}};

      …

public void PlacePeg(int row, int column)
{
      pegs[row, column] = "O";
      if (NextPlayerIsX) pegs[row, column] = "X";
}

public bool PegAtPositionIsX(int row, int column)
{
      return pegs[row, column] == "X";
}
```

Now, let's move on to the next test, namely "Place peg at occupied position." Here is the test:

```
[Test,
      ExpectedException(typeof(TickTackToeBoardException))]
public void testPlacePegAtOccupiedPosition()
{
      board.FirstPlayerPegIsX = true;
      board.PlacePeg(0, 1);
      board.PlacePeg(0, 1);
}
```

The success of this test is in the failure of the code by throwing an exception. So, we declare an attribute ExpectedException which tells NUnit to make sure that an exception of type TickTackToeBoardException is being thrown. You may also ask for it to verify that a specific message has been thrown as part of the exception.

As I am writing this test, two thoughts come to mind. What about setting the first peg without setting the first player? Second, what about getting the peg from an unoccupied position? At this moment, I should avoid the urge to implement these tests, but enter them into the task list as shown below:

Task List

1. ~~Create board~~
2. ~~Set First player~~
3. ~~Set First player again~~
4. Set First Player after game starts
5. ~~Place first peg~~
6. Place peg at occupied position (being implemented right now)
7. Place peg out of column range
8. Place peg out of row range
9. Place Peg without setting first player
10. Get peg from unoccupied position

Now, let's complete the test for task #6. We need to implement the code for that as shown below:

```
using System;

namespace TickTackToeLib
{
	public class TickTackToeBoardException : ApplicationException
	{
		public TickTackToeBoardException(string message)
			: base(message)
		{
		}
	}
}
```

And in the TickTackToeBoard class, we modify the PlacePeg method as follows:

```
		public void PlacePeg(int row, int column)
		{
			if (pegs[row, column] != String.Empty)
			{
				throw new TickTackToeBoardException(
							"Position occupied");
			}

			pegs[row, column] = "O";
			if (NextPlayerIsX) pegs[row, column] = "X";
		}
```

With this the bar goes from red to green. Let's go ahead and implement the remaining tests here and we add two more tests while at it to test range for getting peg at position. We are showing both the tests and the supporting code here:

Place Peg out of column range:

```
[Test,
      ExpectedException(typeof(TickTackToeBoardException))]
public void testPlacePegOutOfColumnRange()
{
      board.FirstPlayerPegIsX = false;
      board.PlacePeg(1, 3);
}

public void PlacePeg(int row, int column)
{
      if(row < 0 || row > 2)
            throw new TickTackToeBoardException(
                  "Row out of range");

      if (column < 0 || column > 2)
            throw new TickTackToeBoardException(
                  "Column out of range");

      if (pegs[row, column] != String.Empty)
      {
            throw new TickTackToeBoardException(
                  "Position occupied");
      }

      pegs[row, column] = "O";
      if (NextPlayerIsX) pegs[row, column] = "X";
}
```

Place peg out of row range:

```
[Test,
      ExpectedException(typeof(TickTackToeBoardException))]
public void testPlacePegOutOfRowRange()
{
      board.FirstPlayerPegIsX = false;
      board.PlacePeg(-1, 1);
}
```

Get peg from unoccupied position

```
[Test,
      ExpectedException(typeof(TickTackToeBoardException))]
public void testGetPegFromUnoccupiedPosition()
{
      board.FirstPlayerPegIsX = false;
      board.PegAtPositionIsX(0, 1);
}

public bool PegAtPositionIsX(int row, int column)
{
      if (pegs[row, column] == "")
            throw new TickTackToeBoardException(
                  "Position empty");
```

```
        return pegs[row, column] == "X";
    }
```

Get peg out of column range

```
    [Test,
        ExpectedException(typeof(TickTackToeBoardException))]
    public void testGetPegOutOfColumnRange()
    {
        board.FirstPlayerPegIsX = false;
        board.PegAtPositionIsX(0, 3);
    }

    public bool PegAtPositionIsX(int row, int column)
    {
        if(row < 0 || row > 2)
            throw new TickTackToeBoardException(
                "Row out of range");

        if (column < 0 || column > 2)
            throw new TickTackToeBoardException(
                "Column out of range");

        if (pegs[row, column] == "")
            throw new TickTackToeBoardException(
                "Position empty");

        return pegs[row, column] == "X";
    }
```

Now that we have a green bar, good time to refactor[4] some code that is violating the DRY[6] principle.

```
    private void CheckRange(int row, int column)
    {
        if(row < 0 || row > 2)
            throw new TickTackToeBoardException(
                "Row out of range");

        if (column < 0 || column > 2)
            throw new TickTackToeBoardException(
                "Column out of range");
    }

    public void PlacePeg(int row, int column)
    {
        CheckRange(row, column);

        if (pegs[row, column] != String.Empty)
        {
            throw new TickTackToeBoardException(
                "Position occupied");
        }
```

```
            pegs[row, column] = "O";
            if (NextPlayerIsX) pegs[row, column] = "X";
        }

        public bool PegAtPositionIsX(int row, int column)
        {
            CheckRange(row, column);

            if (pegs[row, column] == "")
                throw new TickTackToeBoardException(
                            "Position empty");

            return pegs[row, column] == "X";
        }
```

Get peg out of row range

```
        [Test,
            ExpectedException(typeof(TickTackToeBoardException))]
        public void testGetPegOutOfRowRange()
        {
            board.FirstPlayerPegIsX = false;
            board.PegAtPositionIsX(-2, 1);
        }
```

Set First Player after game starts

```
        [Test,
            ExpectedException(typeof(TickTackToeBoardException))]
        public void testSetFirstPlayerAfterGameBeging()
        {
            board.FirstPlayerPegIsX = false;
            board.PlacePeg(0, 1);
            board.FirstPlayerPegIsX = true;
        }

    public class TickTackToeBoard
    {
        …
        private bool gameStarted = false;

        public bool FirstPlayerPegIsX
        {
            get { return NextPlayerIsX; }
            set
            {
                if (gameStarted)
                    throw new TickTackToeBoardException(
                        "Game has begun");

                NextPlayerIsX = value;
            }
        }

        public void PlacePeg(int row, int column)
```

```
        {
            CheckRange(row, column);

            if (pegs[row, column] != String.Empty)
            {
                throw new TickTackToeBoardException(
                    "Position occupied");
            }

            pegs[row, column] = "O";
            if (NextPlayerIsX) pegs[row, column] = "X";

            gameStarted = true;
        }

        …
    }
```

Place Peg without setting first player

```
        [Test,
            ExpectedException(typeof(TickTackToeBoardException))]
        public void testPlacePegWithoutSettingFirstPlayer()
        {
            board.PlacePeg(0, 1);
        }

    public class TickTackToeBoard
    {
        …
        private bool firstPlayerSet = false;

        public bool FirstPlayerPegIsX
        {
            get { return NextPlayerIsX; }
            set
            {
                if (gameStarted)
                    throw new TickTackToeBoardException(
                    "Game has begun");

                NextPlayerIsX = value;
                firstPlayerSet = true;
            }
        }

        public void PlacePeg(int row, int column)
        {
            if (!firstPlayerSet)
                throw new TickTackToeBoardException(
                    "First player not set");
            …

        …

    }
```

The task list now looks like this:

Task List

1. ~~Create board~~
2. ~~Set First player~~
3. ~~Set First player again~~
4. ~~Set First Player after game starts~~
5. ~~Place first peg~~
6. ~~Place peg at occupied position~~
7. ~~Place peg out of column range~~
8. ~~Place peg out of row range~~
9. ~~Place Peg without setting first player~~
10. ~~Get peg from unoccupied position~~
11. ~~Get peg out of column range~~
12. ~~Get peg out of row range~~

Its time now to think of more tests to write:

Task List

…

13. Set second Peg
14. Game win through column alignment
15. Game win through row alignment
16. Game win through diagonal alignment
17. Place peg after game win

Let's implement these tests now:

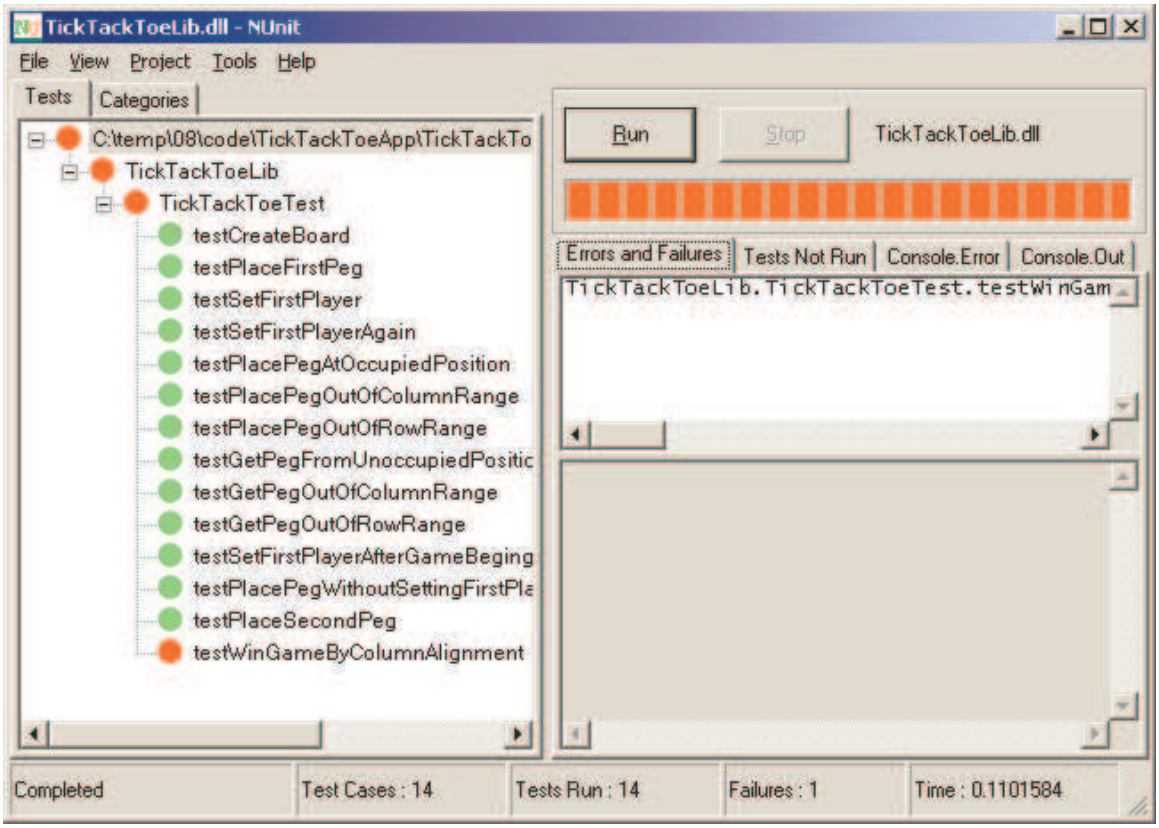
Set second Peg

```
[Test]
public void testPlaceSecondPeg()
{
      board.FirstPlayerPegIsX = false;
      board.PlacePeg(0, 1);
      Assert.IsFalse(board.PegAtPositionIsX(0, 1));
      board.PlacePeg(1, 2);
      Assert.IsTrue(board.PegAtPositionIsX(1, 2));
}

public void PlacePeg(int row, int column)
{
      … (not shown)

      NextPlayerIsX = !NextPlayerIsX;

      gameStarted = true;
}
```

Game win through column alignment

```
[Test]
public void testWinGameByColumnAlignment()
{
	board.FirstPlayerPegIsX = false;
	board.PlacePeg(0, 0); // "O"
	board.PlacePeg(1, 2);
	board.PlacePeg(1, 0); // "O"
	board.PlacePeg(2, 2);
	board.PlacePeg(2, 0); // "O"
	Assert.IsTrue(board.GameOver);
}
```

Now the NUnitGUI displays the failure of this test:



In order to fix this, we need to fix the GameOver property. Looking at the GameOver property, we find that we have lied out way so far at it!

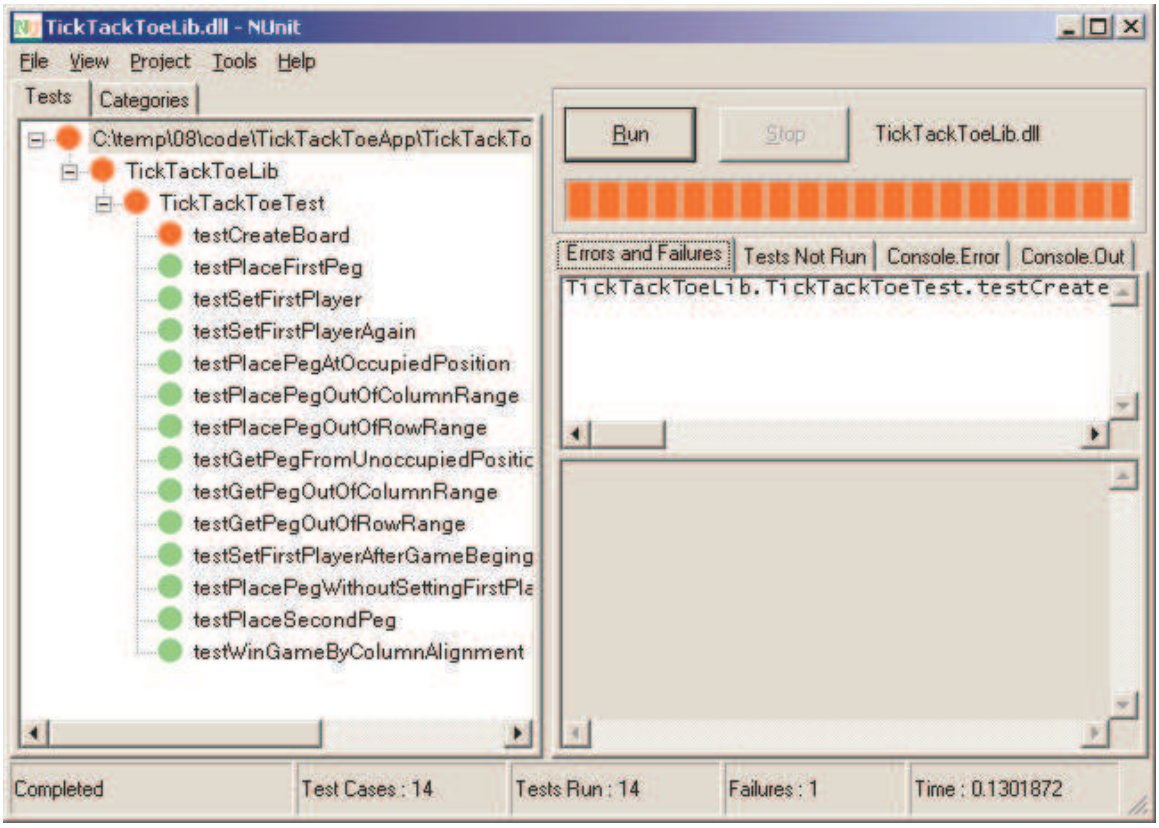
```
public bool GameOver
{
	get { return false; }
}
```

Let's fix this:

```
public bool GameOver
```

```
{
        get
        {
                return CheckColumnAlignment();
        }
}

private bool CheckColumnAlignment()
{
        bool result = false;
        for(int i = 0; i < 3; i++)
        {
                if (pegs[0, i] == pegs[1, i]
                        && pegs[1, i] == pegs[2, i])
                {
                        result = true;
                        break;
                }
        }

        return result;
}
```

Now running NUnitGui we see the following:



Incidentally, while the last test we are writing succeeds, the very first test is failing! The game is over even before we played! ***Test cases are our angels.*** They let us refactor and

evolve the code, giving us the confidence that they are there around watching out for us. See how the problem in the GetColumnAlignment came to surface pretty quickly. Let's fix this now:

```
…
if (pegs[0, i] == pegs[1, i]
&& pegs[1, i] == pegs[2, i] &&
      pegs[2, i] != String.Empty)
…
```

Game win through row alignment

```
[Test]
public void testWinGameByRowAlignment()
{
      board.FirstPlayerPegIsX = true;
      board.PlacePeg(1, 0); // "X"
      board.PlacePeg(0, 2);
      board.PlacePeg(1, 2); // "X"
      board.PlacePeg(2, 0);
      board.PlacePeg(1, 1); // "X"
      Assert.IsTrue(board.GameOver);
}
```

```
public bool GameOver
{
      get
      {
            return CheckColumnAlignment()
            || CheckRowAlignment();
      }
}

private bool CheckRowAlignment()
{
      bool result = false;
      for(int i = 0; i < 3; i++)
      {
            if (pegs[i, 0] == pegs[i, 1] && pegs[i, 1]
            == pegs[i, 2] &&
                  pegs[i, 2] != String.Empty)
            {
                  result = true;
                  break;
            }
      }

      return result;
}
```

Game win through diagonal alignment

```
[Test]
public void testWinGameByDiagonalAlignment()
```

```
{
	board.FirstPlayerPegIsX = true;
	board.PlacePeg(0, 2); // "X"
	board.PlacePeg(0, 0);
	board.PlacePeg(1, 1); // "X"
	board.PlacePeg(2, 2);
	board.PlacePeg(2, 0); // "X"
	Assert.IsTrue(board.GameOver);
}

public bool GameOver
{
	get
	{
		return CheckDiagonalAlignment()
			|| CheckColumnAlignment()
			|| CheckRowAlignment();
	}
}

private bool CheckDiagonalAlignment()
{
	bool result = false;

	if (pegs[0, 0] == pegs[1, 1] && pegs[1, 1]
		== pegs[2, 2]
		&& pegs[2, 2] != String.Empty)
	{
		result = true;
	}
	else
	{
		if (pegs[0, 2] == pegs[1, 1] && pegs[1, 1]
			== pegs[2, 0]
			&& pegs[2, 0] != String.Empty)
		{
			result = true;
		}
	}

	return result;
}
```

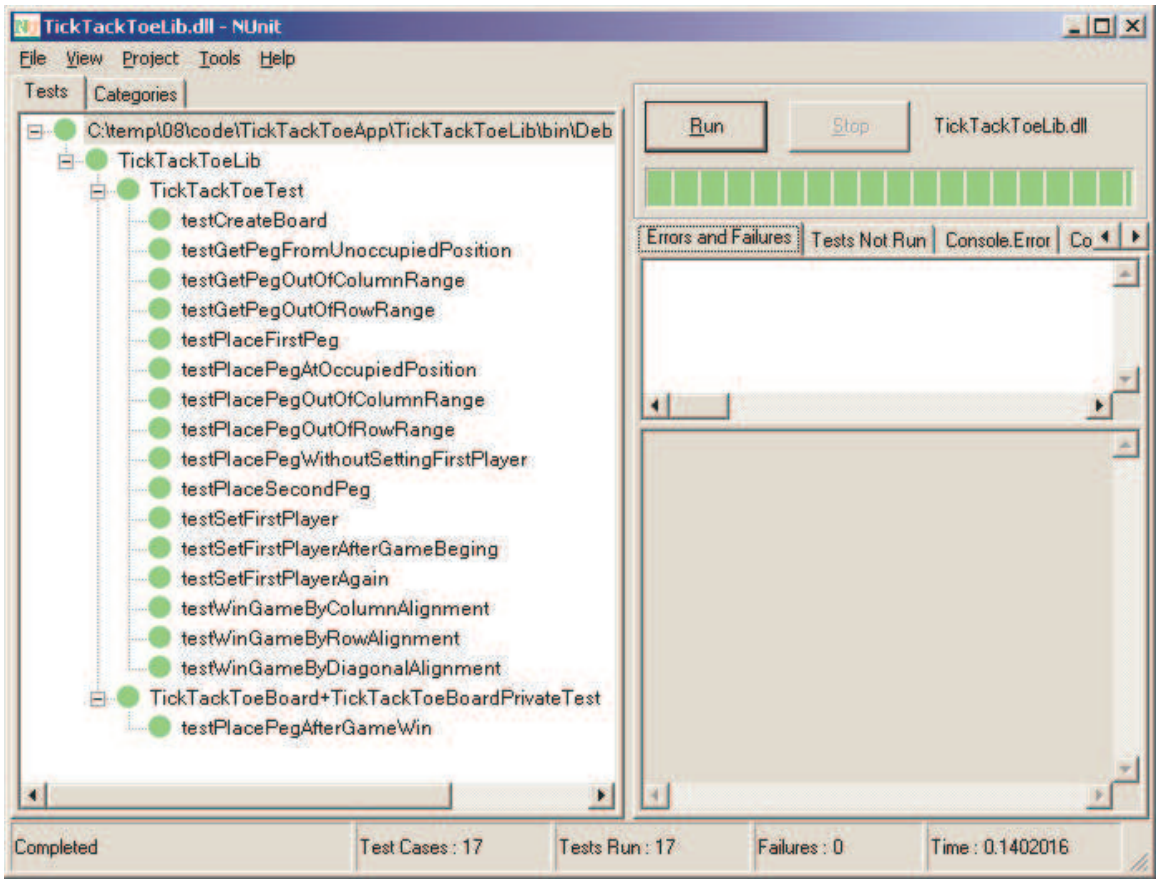
Place peg after game win

I want to test the behavior if a peg is placed after game is over. I want to set the condition for the game to be over directly instead of going through the public interface. How can I do that? Well, how about writing the test in a nested class?

```
[TestFixture]
public class TickTackToeBoardPrivateTest
{
	private TickTackToeBoard board;

	[SetUp]
```

```
        public void createBoard()
        {
              board = new TickTackToeBoard();
        }

        [Test,
        ExpectedException(typeof(TickTackToeBoardException))]
        public void testPlacePegAfterGameWin()
        {
              board.pegs[0, 0] = board.pegs[1, 1] =
                    board.pegs[2, 2] = "X";
              board.PlacePeg(1, 2);
        }
}
```

When I run NUnitGui, I get the following:

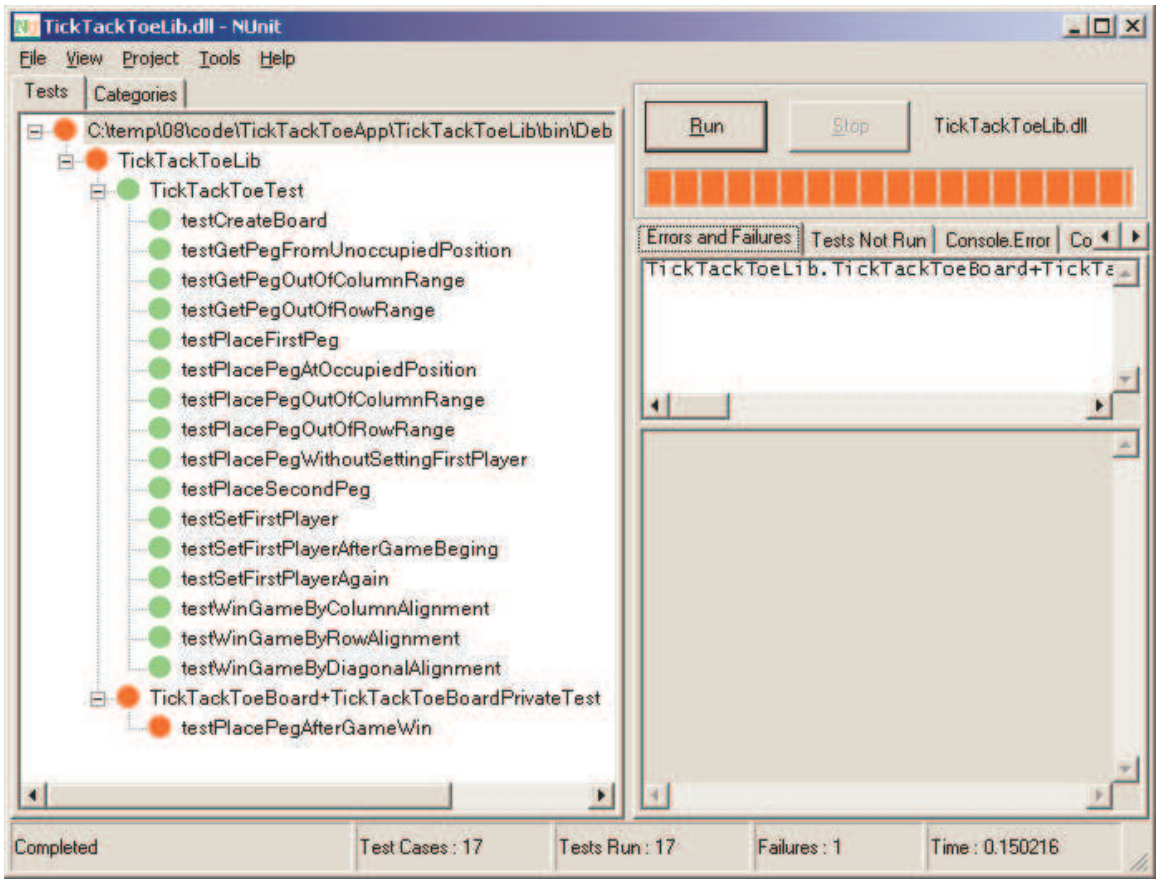


Note that the Nested class shows up at the bottom and wait a minute, all test cases have succeeded. Hummm? I was expecting the most recent test; the one to test placing of a peg after game has been won to fail. But it succeeds. That's great; let's move on, right? No, it is important for a test to fail before it succeeds. The reason it may be succeeding is due to some other failure and not the one you expected. In deed that is the case in this example:

Placing a break point within the testPlacePegAfterGameWin method and debugging through the execution of NUnit, we fill find that the reason for success is that the TickTackToeException is being thrown for the wrong reason – the first player has not been set. Let's fix that in the test and run the NUnitGui again.

```
[Test,
ExpectedException(typeof(TickTackToeBoardException))]
public void testPlacePegAfterGameWin()
{
      board.FirstPlayerPegIsX = true;
      board.pegs[0, 0] = board.pegs[1, 1]
            = board.pegs[2, 2] = "X";
      board.PlacePeg(1, 2);
}
```

Now running the NUnitGui, we find that the test case fails:



That is good. Now we can write the code necessary for this test to succeed.

```
public void PlacePeg(int row, int column) {
      if (GameOver)
            throw new TickTackToeBoardException(
                  "Game already over");
                  …
```

The NUnitGui shows that all tests are now running well. Our task list looks like this now:

Task List

1. ~~Create board~~
2. ~~Set First player~~
3. ~~Set First player again~~
4. ~~Set First Player after game starts~~
5. ~~Place first peg~~
6. ~~Place peg at occupied position~~
7. ~~Place peg out of column range~~
8. ~~Place peg out of row range~~
9. ~~Place Peg without setting first player~~
10. ~~Get peg from unoccupied position~~
11. ~~Get peg out of column range~~
12. ~~Get peg out of row range~~
13. ~~Set second Peg~~
14. ~~Game win through column alignment~~
15. ~~Game win through row alignment~~
16. ~~Game win through diagonal alignment~~
17. ~~Place peg after game win~~

Can we think of any more tests? It is time to move on to developing the user interface. Let's take a look at the entire code for the Test and for the TickTackToeBoard as well:

```
using System;
using NUnit.Framework;

namespace TickTackToeLib
{
	[TestFixture]
	public class TickTackToeTest
	{
		private TickTackToeBoard board;

		[SetUp]
		public void createBoard()
		{
			board = new TickTackToeBoard();
		}

		[Test]
		public void testCreateBoard()
		{
			Assert.IsNotNull(board);
			Assert.IsFalse(board.GameOver);
		}

		[Test]
		public void testSetFirstPlayer()
		{
```

```
        board.FirstPlayerPegIsX = true;
        Assert.IsTrue(board.FirstPlayerPegIsX);
    }

    [Test]
    public void testSetFirstPlayerAgain()
    {
        board.FirstPlayerPegIsX = true;
        Assert.IsTrue(board.FirstPlayerPegIsX);
        board.FirstPlayerPegIsX = false;
        Assert.IsFalse(board.FirstPlayerPegIsX);
    }

    [Test]
    public void testPlaceFirstPeg()
    {
        board.FirstPlayerPegIsX = true;
        board.PlacePeg(0, 1);
        Assert.IsTrue(board.PegAtPositionIsX(0, 1));
    }

    [Test,
    ExpectedException(typeof(TickTackToeBoardException))]
    public void testPlacePegAtOccupiedPosition()
    {
        board.FirstPlayerPegIsX = true;
        board.PlacePeg(0, 1);
        board.PlacePeg(0, 1);
    }

    [Test,
    ExpectedException(typeof(TickTackToeBoardException))]
    public void testPlacePegOutOfColumnRange()
    {
        board.FirstPlayerPegIsX = false;
        board.PlacePeg(1, 3);
    }

    [Test,
    ExpectedException(typeof(TickTackToeBoardException))]
    public void testPlacePegOutOfRowRange()
    {
        board.FirstPlayerPegIsX = false;
        board.PlacePeg(-1, 1);
    }

    [Test,
    ExpectedException(typeof(TickTackToeBoardException))]
    public void testGetPegFromUnoccupiedPosition()
    {
        board.FirstPlayerPegIsX = false;
        board.PegAtPositionIsX(0, 1);
    }

    [Test,
    ExpectedException(typeof(TickTackToeBoardException))]
    public void testGetPegOutOfColumnRange()
```

```
{
      board.FirstPlayerPegIsX = false;
      board.PegAtPositionIsX(0, 3);
}

[Test,
ExpectedException(typeof(TickTackToeBoardException))]
public void testGetPegOutOfRowRange()
{
      board.FirstPlayerPegIsX = false;
      board.PegAtPositionIsX(-2, 1);
}

[Test,
ExpectedException(typeof(TickTackToeBoardException))]
public void testSetFirstPlayerAfterGameBeging()
{
      board.FirstPlayerPegIsX = false;
      board.PlacePeg(0, 1);
      board.FirstPlayerPegIsX = true;
}

[Test,
ExpectedException(typeof(TickTackToeBoardException))]
public void testPlacePegWithoutSettingFirstPlayer()
{
      board.PlacePeg(0, 1);
}

[Test]
public void testPlaceSecondPeg()
{
      board.FirstPlayerPegIsX = false;
      board.PlacePeg(0, 1);
      Assert.IsFalse(board.PegAtPositionIsX(0, 1));
      board.PlacePeg(1, 2);
      Assert.IsTrue(board.PegAtPositionIsX(1, 2));
}

[Test]
public void testWinGameByColumnAlignment()
{
      board.FirstPlayerPegIsX = false;
      board.PlacePeg(0, 0); // "O"
      board.PlacePeg(1, 2);
      board.PlacePeg(1, 0); // "O"
      board.PlacePeg(2, 2);
      board.PlacePeg(2, 0); // "O"
      Assert.IsTrue(board.GameOver);
}

[Test]
public void testWinGameByRowAlignment()
{
      board.FirstPlayerPegIsX = true;
      board.PlacePeg(1, 0); // "X"
      board.PlacePeg(0, 2);
```

```
                board.PlacePeg(1, 2); // "X"
                board.PlacePeg(2, 0);
                board.PlacePeg(1, 1); // "X"
                Assert.IsTrue(board.GameOver);
            }

            [Test]
            public void testWinGameByDiagonalAlignment()
            {
                board.FirstPlayerPegIsX = true;
                board.PlacePeg(0, 2); // "X"
                board.PlacePeg(0, 0);
                board.PlacePeg(1, 1); // "X"
                board.PlacePeg(2, 2);
                board.PlacePeg(2, 0); // "X"
                Assert.IsTrue(board.GameOver);
            }
        }
}


using System;
using NUnit.Framework;

namespace TickTackToeLib
{
        public class TickTackToeBoard
        {
            private bool NextPlayerIsX;
            private string[,] pegs
            = new string[,] {{"", "", ""}, {"", "", ""}, {"", "", ""}};
            private bool gameStarted = false;
            private bool firstPlayerSet = false;

            public bool GameOver
            {
                get
                {
                    return CheckDiagonalAlignment()
                        || CheckColumnAlignment()
                        || CheckRowAlignment();
                }
            }

            private bool CheckDiagonalAlignment()
            {
                bool result = false;

                if (pegs[0, 0] == pegs[1, 1] && pegs[1, 1]
                    == pegs[2, 2]
                    && pegs[2, 2] != String.Empty)
                {
                    result = true;
                }
                else
                {
```

```
            if (pegs[0, 2] == pegs[1, 1] && pegs[1, 1]
                  == pegs[2, 0]
                  && pegs[2, 0] != String.Empty)
            {
                  result = true;
            }
      }

      return result;
}

private bool CheckRowAlignment()
{
      bool result = false;
      for(int i = 0; i < 3; i++)
      {
            if (pegs[i, 0] == pegs[i, 1] && pegs[i, 1]
                  == pegs[i, 2] &&
                  pegs[i, 2] != String.Empty)
            {
                  result = true;
                  break;
            }
      }

      return result;
}

private bool CheckColumnAlignment()
{
      bool result = false;
      for(int i = 0; i < 3; i++)
      {
            if (pegs[0, i] == pegs[1, i] && pegs[1, i]
                  == pegs[2, i] &&
                  pegs[2, i] != String.Empty)
            {
                  result = true;
                  break;
            }
      }

      return result;
}

public bool FirstPlayerPegIsX
{
      get { return NextPlayerIsX; }
      set
      {
            if (gameStarted)
                  throw new TickTackToeBoardException(
                        "Game has begun");

            NextPlayerIsX = value;
            firstPlayerSet = true;
      }
```

```
}

private void CheckRange(int row, int column)
{
      if(row < 0 || row > 2)
            throw new TickTackToeBoardException(
            "Row out of range");

      if (column < 0 || column > 2)
            throw new TickTackToeBoardException(
            "Column out of range");
}

public void PlacePeg(int row, int column)
{
      if (GameOver)
            throw new TickTackToeBoardException(
            "Game already over");

      if (!firstPlayerSet)
            throw new TickTackToeBoardException(
            "First player not set");

      CheckRange(row, column);

      if (pegs[row, column] != String.Empty)
      {
            throw new TickTackToeBoardException(
            "Position occupied");
      }

      pegs[row, column] = "O";
      if (NextPlayerIsX) pegs[row, column] = "X";

      NextPlayerIsX = !NextPlayerIsX;

      gameStarted = true;
}

public bool PegAtPositionIsX(int row, int column)
{
      CheckRange(row, column);

      if (pegs[row, column] == "")
            throw new TickTackToeBoardException(
            "Position empty");

      return pegs[row, column] == "X";
}

[TestFixture]
public class TickTackToeBoardPrivateTest
{
      private TickTackToeBoard board;

      [SetUp]
      public void createBoard()
```

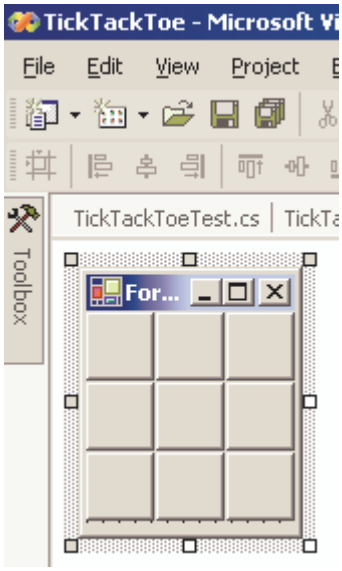
```
                {
                        board = new TickTackToeBoard();
                }

                [Test,
                ExpectedException(typeof(TickTackToeBoardException))]
                public void testPlacePegAfterGameWin()
                {
                        board.FirstPlayerPegIsX = true;
                        board.pegs[0, 0] = board.pegs[1, 1]
                                = board.pegs[2, 2] = "X";
                        board.PlacePeg(1, 2);
                }
        }
    }
}
```

The UI

The interesting part of this exercise is that the middle tier business logic is pretty solid. It is as robust as it could be (assuming our tests are fairly complete). And we achieved that without writing any UI code! Typically, the above exercise (building the library) takes about one hour for this example, while discussing the issues with a group of seven to fifteen attendees.

It took only 8 minutes to build the UI from scratch! Here is the UI:

We will create a windows application project named TickTackToe with in the current blank solution. In the Form class, we will place button as shown here:

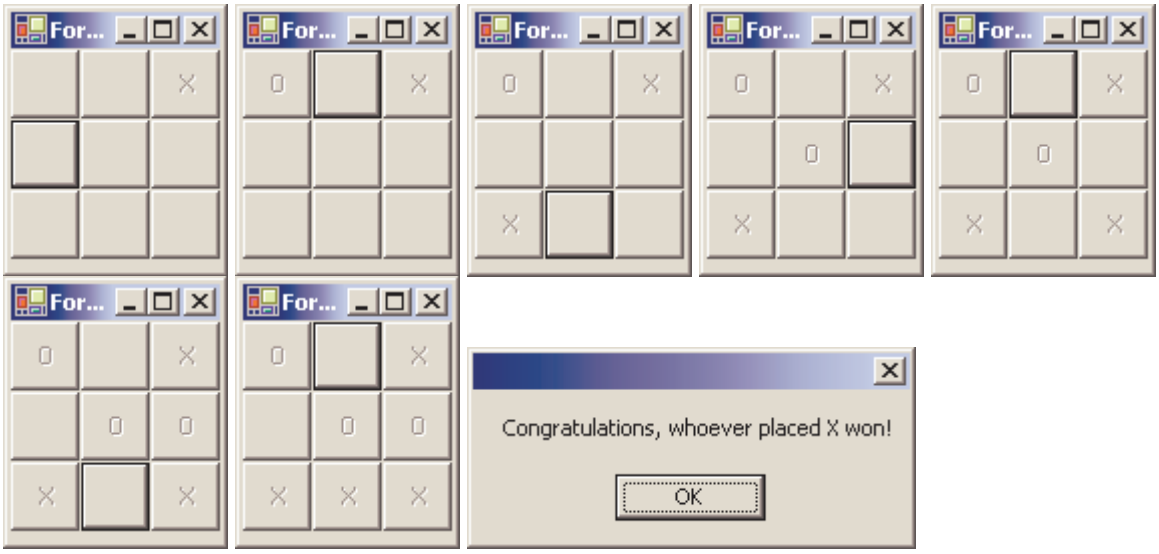


Nine buttons have been placed. We will give names for the buttons as button_R_C where R is the row (0, 1 or 2) and C is the column (0, 1 or 2). This will allow us to write an handler that can be used to process the click events on the buttons as shown below:

```
private TickTackToeBoard board = new TickTackToeBoard();

private void Form1_Load(object sender, System.EventArgs e)
{
      button_0_0.Click
            += new EventHandler(HandleButtonEvent);
      button_0_1.Click
            += new EventHandler(HandleButtonEvent);
      button_0_2.Click
            += new EventHandler(HandleButtonEvent);

      button_1_0.Click
            += new EventHandler(HandleButtonEvent);
      button_1_1.Click
            += new EventHandler(HandleButtonEvent);
      button_1_2.Click
            += new EventHandler(HandleButtonEvent);

      button_2_0.Click
            += new EventHandler(HandleButtonEvent);
      button_2_1.Click
            += new EventHandler(HandleButtonEvent);
      button_2_2.Click
            += new EventHandler(HandleButtonEvent);

      board.FirstPlayerPegIsX = true;
      //Setting first player to be X.
      // You can easily implement
      // a menu to modify this. Not shown here.
}

private void HandleButtonEvent(object sender, EventArgs e)
{
      try
      {
            if(!board.GameOver)
            {
                  int row;
                  int column;
                  Button theButton = sender as Button;
                  string name = theButton.Name;

                  row = Convert.ToInt32(
                        name.Split(new char[]{'_'})[1]);
                  column = Convert.ToInt32(
                        name.Split(new char[]{'_'})[2]);

                  board.PlacePeg(row, column);
                  theButton.Text = "O";
                  if (board.PegAtPositionIsX(row, column))
                        theButton.Text = "X";
                  theButton.Enabled = false;

                  if (board.GameOver)
                  {
                        MessageBox.Show(
                  "Congratulations, whoever placed " +
```

```
                                        theButton.Text + " won!");
                        }
                }
        }
        catch(Exception ex)
        {
                MessageBox.Show(ex.Message);
        }
}
```

The game in action:



Advantage of Layering

I was teaching a course recently and went through this exercise at which point an enthusiastic set of attendees asked me to put together an ASP.NET application to play tick tack toe. Even though it may not be best to develop it as a web application, to illustrate the advantage, we threw in an ASP.NET application. We moved around just a little bit from the Windows UI into another middle tier Utility class to avoid code duplication and within about five minutes we had ASP.NET version of tick tack toe running, with no change to the TickTackToeLib. I can't over emphasize the benefit of good separation and layering. What's interesting is that TDD (and TFC) naturally leads us into that.

Well, what about keeping the score

In this iteration we did not cover that. We will do that in the next iteration and that's a good place to talk about Mock objects. That will be discussed in Part II.

Benefits from TFC

There are several benefits that one could see from an approach like this:

- It simplifies the design
- Completely revert the way we develop

- Makes us think about how our object would be used
- Helps us develop better interfaces that are easier to use
- Would change the way we perceive things
- Makes the code easily testable
- It decouples our code from its surroundings
- Serves as invaluable form of documentation
- Naturally partitions the system into layers
- Makes the code robust
- It gives an good opportunity for us to think of the failures and what we need to accommodate
- It provides a safety net as we refactor the code – ***the test cases are our angels***
- It simply boosts the confidence in our code

Principles followed so far

We followed a few principles in this example: YAGNI and DRY principle. We will see how other principles come into play in Part II.

General Guidelines

OK, we wrote our tests and every thing is working great. Then some one draws attention to a bug in the code. Of course we feel pretty embarrassed when that happens and want to fix the bug right away. But wait, do not fix the bug. First write a test code that will fail because of the bug. Then fix the bug so that the test will now succeed. The advantage of this is if some other change, modification or addition to the code results in the same bug creeping in later, the test angel will catch that.

Also, I ran into this situation on a project and learnt this lesson. I had over fifty test cases for my code running successfully before I gave the code for integration. The person integrating was having problems. Each time he said the code does not work, all that I had to do was show him the relevant test code. After a couple of hours and several calls to "no it does not work," I looked at one situation and I was some what puzzled. The way he was accessing my code looked pretty reasonable and it should work, but did not. I though, "Hum, did I forget to write a test for that?" When I opened the project and looked at the tests, I found that I did have a test for it! So, what is wrong? Eventually we figured out that I was running on Windows 2003 while he was running on Windows XP. Digging into the MSDN documentation I eventually found that that particular functionality I was using worked differently on these two versions of windows. Lesson learnt: Run your unit tests on all supported platforms. But that could get tedious, isn't it? No, not if we use continuous integration as we will see in Part III of this article.

Here are some things that we need to keep in mind while writing out test cases:

- Red/Green/Refactor should be our mantra
- Stay one step away from Green at all times – do not make many changes at once
- Place the tests near the code (in same project, some tests as nested as well)
- Isolate your tests – failure of one should not affect the other
- Write with assert in mind

- Write a test for a bug you find
- Do not refactor code without having test cases to support
- Test on all your platforms

What's next?

In this Part I we focused on Test First Coding. In Part II we will take a close look at Mock objects and some frameworks for that. In Part III we will talk about continuous integration and tools available for that as well.

Conclusion

Unit testing is a way of design than verification. It simplifies our design. It makes the code more testable. It improves robustness of code and our confidence. NUnit makes it very easy to utilize the principles of Test Driven Development. The use of these techniques and tools is gaining good momentum and I hope you will put these to good use on your projects as well.

Your feedback

Tell it like it is. Did you like the article; was it useful, do you want to see more such articles? Let us know, as that will motivate us to continue writing. Did you not like it? Please tell us so we can improve on it. Your constructive criticism makes a difference. Do you have suggestions for improvement? Please send those to use and we will consider incorporating those.

References

1. "Test-Driven Development By Example," Ken Beck, Addison-Wesley.
2. "Test-Driven Development in Microsoft .NET," James W. Newkirk, Alexei A. Vorontsov.
3. "Agile Software Development, Principles, Practices and Patterns," Robert C. Martin, Prentice Hall.
4. "Refactoring Improving The Design Of Existing Code," Martin Fowler, Addison-Wesley.
5. "NUnit 2.2," (NUnit-2.2.0.msi) at http://www.sourceforge.net/projects/nunit.
6. "Pragmatic Programmer – From Journeyman to Master," Andy Hunt, Dave Thomas, Addison-Wesley.