Test Driven Development – Part II: Mock Objects

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

Abstract

In this second of the three part series on Test Driven Development, we focus on using Mock objects to isolate our code from its dependencies so as to make it testable and also to further development when the dependent components are not quite ready or available. In this article we discuss the benefits of Mock objects, show how to write one and finally present an example that uses the EasyMock framework. In Part III we will look at continuous integration.

Where are we?

In Part I we discussed the benefits of using NUnit by going through the first iteration of building a simple application – the Ticktacktoe game. Where we left off was development of testable business logic with sixteen test cases and the TickTackToeBoard class. It was developed with Test First approach and we discussed the benefits and how the act of test first development is more of a design than mere testing or verification. So much that the word "Test" in Test First Development or Test Driven Development is some what misleading. We assume that you have read the Part I in which we have written the test cases and then the code to implement the logic. What's next?

From where we left, we are looking at implementing the facility to store the scores of winners. How are we going to store the scores? We could store it in a database. How about XML, raw text file? Oh no, we have to use a web service for that right☺. What if we want to have the flexibility of changing the way we store the information. We may not want to implement different solutions right now, however, having one solution fully engrained into our code so much that it will be hard to change it later on is not the smartest thing to do, isn't it?

Isolating the storage

Let's start out by looking at ways we can isolate the storage. Instead of our code depending on the specific database calls, etc., we will rely on another class that will take care of that storage. Our code will depend on an interface so the implementation can be modified or replaced easily. This is based on the Dependency Inversion Principle[1] as shown below:

Let' start with the test first approach in implementing the storage of scores.

Reviewing the code and deciding the next test

The classes we have so far are the following:
TickTackToeTest – Test cases for the TickTackToe class
TickTackToeBoard – The business class that models the rules and logic of the game
TickTackToeBoardException – Exception class used by TickTackToeBoard
Form1 – The UI class for the game

So, we want to write a test to store the score right? Where should we put it? The first choice is to put it in TickTackToeTest class. However, since we are starting out testing a different functionality and given the fact that the TickTackToeTest class already has close to 20 methods in it, it will be better to write it as a separate class.

Test for Score Store

We will create a class called ScoreStoreTest and write a method testSetScore.

```
using System;
using NUnit.Framework;

namespace TickTackToeLib
{
	[TestFixture]
	public class ScoreStoreTest
	{
		[Test]
		public void testSetScore()
		{

		}
	}
}
```

Now, it is time to write the test code. What should we write? How about the following:

```
	string PLAYER = "Venkat";
	TickTackToeBoard board = new TickTackToeBoard();
	int score = board.GetScore(PLAYER);
	board.UpdateScore(PLAYER);
	Assert.AreEqual(score + 1, board.GetScore(PLAYER);
```

Well, we are asking the TickTackToeBoard to get the score for the player. Then we ask it to update the score for the player (increase it by one). Looks reasonable? Almost! The role of TickTackToeBoard (from code written so far) is clearly to manage the game rules on the board. Now we are asking it to manage the score as well. This additional responsibility will result in reducing the cohesiveness of this class. We will violate the Single Responsibility Principle (SRP)[3].

It is better to relegate this task to a new class whose sole responsibility is to deal with scores. So, here is the modified test code:

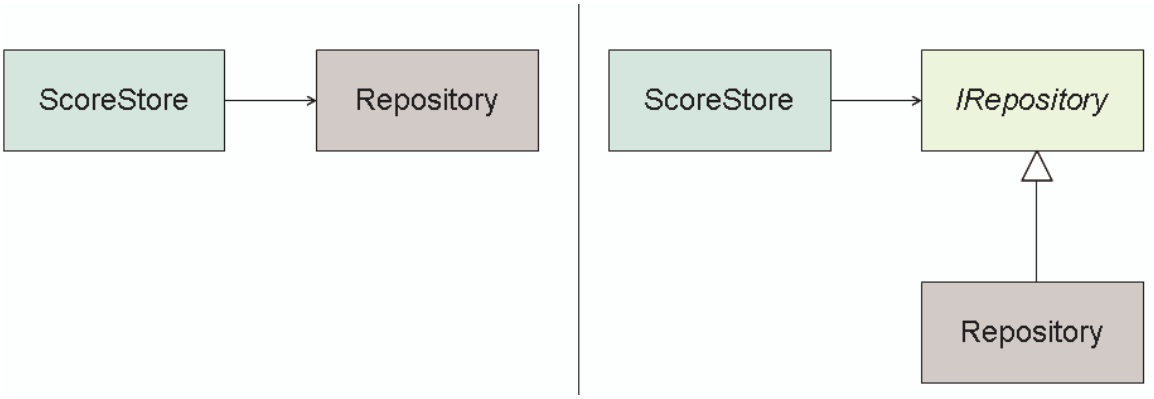
```
	string PLAYER = "Venkat";
	ScoreStore theScoreStore = new ScoreStore();
	int score = theScoreStore.GetScore(PLAYER);
	theScoreStore.UpdateScore(PLAYER);
	Assert.AreEqual(score + 1,
		theScoreStore.GetScore(PLAYER));
```

Now, how do we implement the ScoreStore class?

Implementing Score Store

Well, should we use a database? How about storing the data in an XML file or simply a text file? Do we want to tie ourselves to a particular implementation now? Do we want to be able to have the flexibility of easily modifying the storage medium or means without affecting much code? Further, if we go the route of using a database, we need to ask questions like what DBMS to use, how to organize the tables, what is the data model, etc.

We certainly want to make our code testable without being dependent on a database or a server. In other words, we want isolation. One way to realize this is using what is called Inversion of Control. This is also the concept we learn from Dependency Inversion Principle. Instead of depending on the implementation, our code depends on an interface. At runtime, an implementation of that interface "binds" to the interface reference we depend upon. The following diagram illustrates this concept.



On the left we see the ScoreStore class depend on a class that provides a repository for storage. This may use a database, XML, etc. to actually store the information. On the right we show how the ScoreStore now depends on an interface instead of the concrete class. This "inversion of control" or "inversion of dependency" gives us the flexibility to replace the actual repository (using some kind of a factory if desired) without affecting the rest of the code.

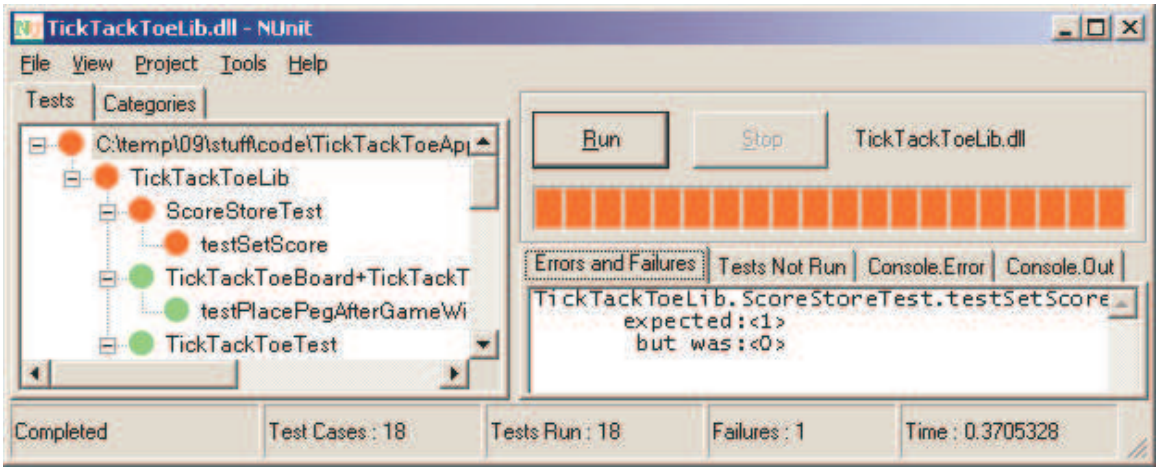
Can we see the ScoreStore code?

OK, we can. Oh well, sorry, we do not have it. Let's write it now.

```
using System;

namespace TickTackToeLib
{
	public class ScoreStore
	{
		public int GetScore(string player)
		{
			return 0;
		}

		public void UpdateScore(string player)
		{
		}
```

```
        }
}
```

If we run NUnit now, we get the following:



That's great we have a test that fails first! Now it is a question of implementing the code for GetScore and UpdateScore. Let's start with GetScore.

Hum, what can we do? Well, we agreed that we will depend on the IRepository. So, let use that.

```
public int GetScore(string player)
{
      try
      {
            return theRepository.GetScore(player);
      }
      catch(ApplicationException)
      {
            return 0;
      }
}
```

And then of course the UpdateScore method needs to be written. Let's see how we can write that:

```
public void UpdateScore(string player)
{
      int score = GetScore(player);
      theRepository.SetScore(player, score + 1);
}
```

Looks reasonable? Well, sure, but where in the world did we get that "theRepository" from? Hum. Well, let's ask the creator of ScoreStore to send it to us. How about that? So, let's write a constructor for the ScoreStore and add a field:

```
        private IRepository theRepository;

        public ScoreStore(IRepository repository)
        {
            theRepository = repository;
        }
```

Of course we need to define the IRepository interface and here we have it:

```
using System;

namespace TickTackToeLib
{
    public interface IRepository
    {
        /// <summary>
        /// Returns the score for the player
        /// </summary>
        /// <param name="player">
        /// player whose score is expected</param>
        /// <returns>Score if present</returns>
        /// <exception cref="ApplicationException">
        /// If player not found</exception>
        /// <exception cref="RepositoryException">
        /// Error accessing the repository
        /// </exception>
        int GetScore(string player);

        /// <summary>
        /// Set the score for the player.
        /// Creates entry for player if not present.
        /// </summary>
        /// <param name="player">
        /// player whose score is expected</param>
        /// <param name="score">The score to set</param>
        /// <exception cref="RepositoryException">
        /// Error accessing the repository
        /// </exception>
        void SetScore(string player, int score);
    }
}
```

Now, if we compile, we should get one error on the test code that the ScoreStore constructor requires an argument. After a quick fix to get the compilation succeed we have:

```
    [Test]
    public void testSetScore()
    {
        string PLAYER = "Venkat";
        ScoreStore theScoreStore = new ScoreStore(null);
        int score = theScoreStore.GetScore(PLAYER);
        theScoreStore.UpdateScore(PLAYER);
        Assert.AreEqual(score + 1,
            theScoreStore.GetScore(PLAYER));
```

```
	}
```

OK, it is time to get the IRepository implemented.

A quick implementation of IRepository

We want to quickly get the functionality of the ScoreStore tested. Further, we do not want to spend great deal of effort, at least at this instance, trying to figure out how to store the data in a database, etc. What is the point in doing that before making sure we have a reasonable understanding of what the ScoreStore should be doing in the first place and what its needs are, right? So, why not create an in-memory repository?

```
using System;
using System.Collections;

namespace TickTackToeLib
{
	public class InMemoryRepository : IRepository
	{
		private Hashtable scores = new Hashtable();

		#region IRepository Members

		public int GetScore(string player)
		{
			if (scores.Contains(player))
				return (int)(scores[player]);
			else
			    throw new ApplicationException("Invalid");
		}

		public void SetScore(string player, int score)
		{
			if (scores.Contains(player))
			{
				scores[player] = (int)(scores[player]) + 1;
			}
			else
			{
				scores[player] = score;
			}
		}

		#endregion
	}
}
```

The InMemoryRespository simply stores the score in the memory. There is no persistent storage. This is enough for us to get the rest of the functionality tested right now, isn't it?

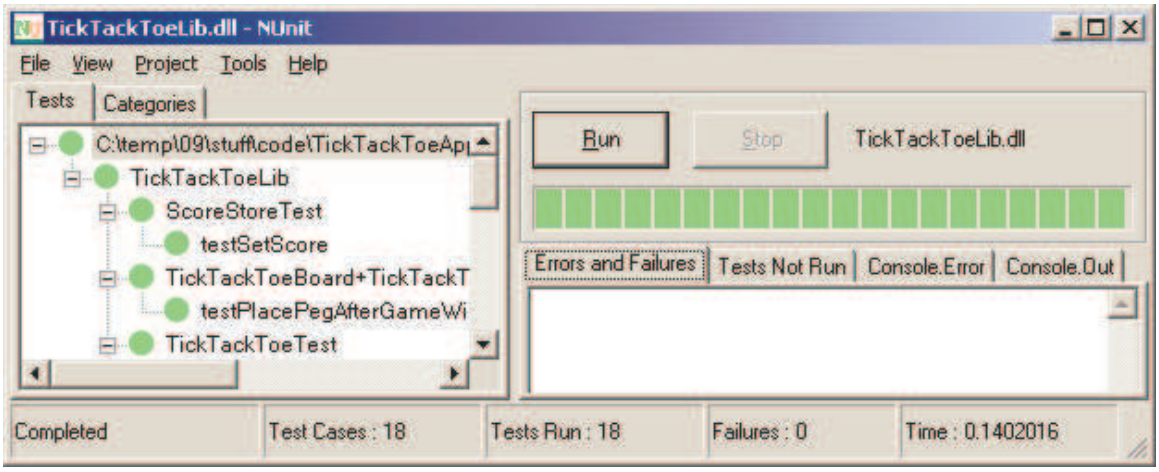
Testing with the InMemoryRepository

Let's modify the test case to work with this mock repository we have created.

```
using System;
```

```
using NUnit.Framework;

namespace TickTackToeLib
{
	[TestFixture]
	public class ScoreStoreTest
	{
		private IRepository theRepository = null;

		protected virtual IRepository createRepository()
		{
			return new InMemoryRepository();
		}

		private IRepository getRepository()
		{
			if (theRepository == null)
			{
				theRepository = createRepository();
			}

			return theRepository;
		}

		[Test]
		public void testSetScore()
		{
			string PLAYER = "Venkat";
			ScoreStore theScoreStore
				= new ScoreStore(getRepository());
			int score = theScoreStore.GetScore(PLAYER);
			theScoreStore.UpdateScore(PLAYER);
			Assert.AreEqual(score + 1,
				theScoreStore.GetScore(PLAYER));
		}
	}
}
```

Running the test now, we get the following output in NUnit:



We also need a way to find the scores of all players. Here is the test for it and the related code.

```
// Par of the ScoreStoreTest class
[Test]
public void testGetScores()
{
      ScoreStore theScoreStore
            = new ScoreStore(getRepository());

      theScoreStore.UpdateScore("PLAYER1");
      theScoreStore.UpdateScore("PLAYER2");

      Assert.IsTrue(theScoreStore.GetScores().Count > 1);
}
```

```
// Part of the ScoreStore class
public System.Collections.Hashtable GetScores()
{
      return theRepository.GetScores();
}
```

```
// Part of the IRepository interface
/// <summary>
/// Returns scores for all players
/// </summary>
/// <returns>Hashtable of player names and score</returns>
/// <exception cref="RepositoryException">
/// Error accessing the repository
/// </exception>
System.Collections.Hashtable GetScores();
```

```
// Part of the InMemoryRepository class
public Hashtable GetScores()
{
      return scores;
}
```

Fixing the GUI to store score

At this point, we can go ahead and fix the GUI to work with this repository and get that functionality tested. So, here is the code change.

We will get started with a dialog to display the scores.

The code for the dialog is given below:

```
private ScoreStore theStore;

public ScoresStatisticsDialog(ScoreStore store)
{
	theStore = store;
	//
	// Required for Windows Form Designer support
	//
	InitializeComponent();
}

private void ScoresStatisticsDialog_Load(object sender,
	System.EventArgs e)
{
	Hashtable scores = theStore.GetScores();

	ArrayList list = new ArrayList();
	foreach(object key in scores.Keys)
	{
		list.Add(key.ToString() + " - " + scores[key]);
	}

	scoresListBox.DataSource = list;
}

private void saveButton_Click(object sender,
	System.EventArgs e)
{
	if (nameTextBox.Text.Trim() != String.Empty)
	{
		theStore.UpdateScore(
			nameTextBox.Text.Trim());
```

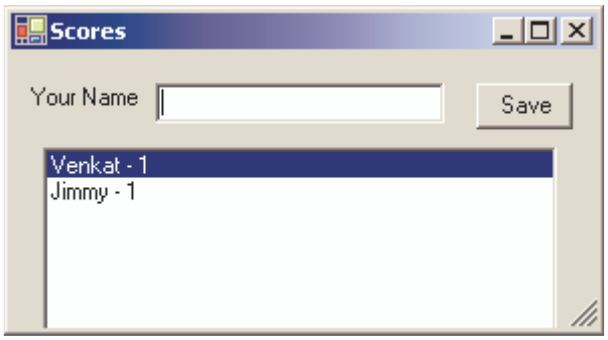
```
                Close();
            }
        }
```

Now change to the windows form to display and record the win:

```
        private bool winnerPegIsX;
        private ScoreStore theStore
            = new ScoreStore(new InMemoryRepository());

        private void HandleButtonEvent(object sender, EventArgs e)
        {
                … // Code not shown …

                    if (board.GameOver)
                    {
                        MessageBox.Show(
                        "Congratulations, whoever placed "
                        + theButton.Text + " won!");

                        RecordWinnerAndDisplayStatistics();
                        winnerPegIsX =
                            board.PegAtPositionIsX(
                                row, column);
                        StartNewGame();
                    }
                }
            }
            catch(Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        private void RecordWinnerAndDisplayStatistics()
        {
            ScoresStatisticsDialog dlg
                = new ScoresStatisticsDialog(theStore);
            dlg.ShowDialog();
        }

        private void StartNewGame()
        {
            // We will let the loser start first
            board = new TickTackToeBoard();
            board.FirstPlayerPegIsX = !winnerPegIsX;
            // There are better ways to do this,
            // but we will leave that as
            // your refactoring exercise :)
            button_0_0.Enabled = true;
            button_0_0.Text = "";
            button_0_1.Enabled = true;
            button_0_1.Text = "";
            button_0_2.Enabled = true;
            button_0_2.Text = "";
```

```
            button_1_0.Enabled = true;
            button_1_0.Text = "";
            button_1_1.Enabled = true;
            button_1_1.Text = "";
            button_1_2.Enabled = true;
            button_1_2.Text = "";

            button_2_0.Enabled = true;
            button_2_0.Text = "";
            button_2_1.Enabled = true;
            button_2_1.Text = "";
            button_2_2.Enabled = true;
            button_2_2.Text = "";
        }
```

After winning a few games, the dialog that displays the winners and scores may look like this:



Well, what about actual storage?

Now that the functionality is in place and we see that it is working, we can now figure out how to actually implement the storage! How about storing the data in XML form? Why not?

How should we write the implementation for the XMLRepository? Test first of course! The code to test the repository is already there. All we want to do is to test it with the XMLRepository. How can we do that? We may want to keep the InMemoryRepository test in tact. Later on as we go though future versions, we still need to isolate and test the functionality without depending on XML repository or any database. So, here is the XMLRespository test.

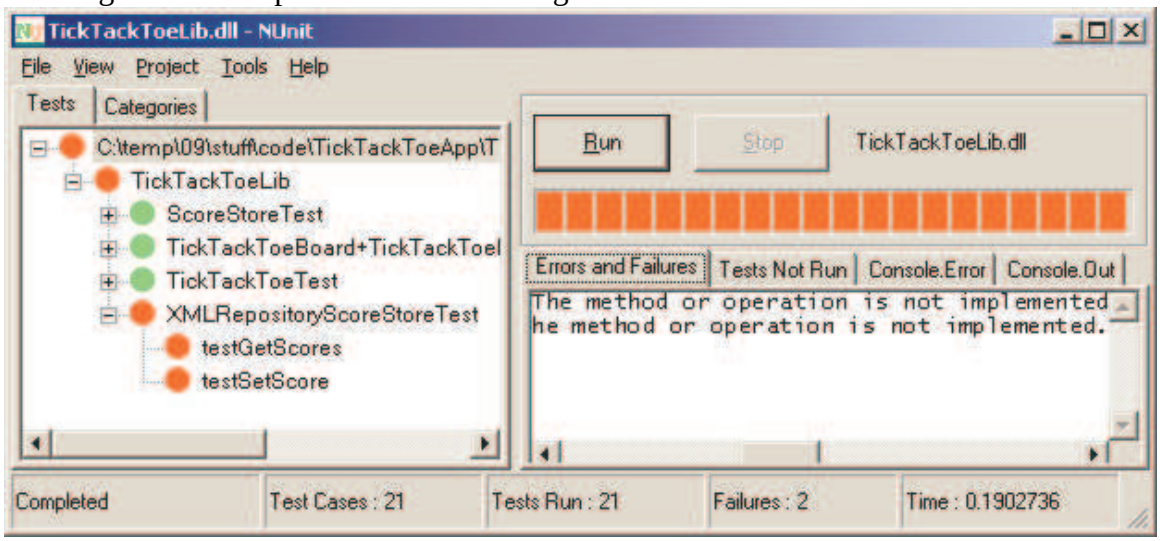
```
using System;

namespace TickTackToeLib
{
	public class XMLRepositoryScoreStoreTest : ScoreStoreTest
	{
		protected override IRepository createRepository()
		{
			return new XMLRepository("myTestXmlFile.xml");
		}
	}
```

```
}
```

Now, we need to implement the XMLRepository. Here is the first step:

```
…
public class XMLRepository : IRepository
{
	private string xmlFileName;

	public XMLRepository(string theRepositoryFile)
	{
		xmlFileName = theRepositoryFile;
	}

	#region IRepository Members

	public int GetScore(string player)
	{
		throw new NotImplementedException();
	}

	public void SetScore(string player, int score)
	{
		throw new NotImplementedException();
	}

	public System.Collections.Hashtable GetScores()
	{
		throw new NotImplementedException();
	}

	#endregion
}
…
```

Running NUnit now produces the following:



How can we implement this now? Well, let's take an easy route. Let's first consider a sample XML file as shown below:

```
<?xml version="1.0" encoding="utf-8" ?>
<!--
This sample XML document is used to
generate schema using xsd. The xsd is run again,
this time on the schema, to generate the
class file scores.cs (xsd /n:TickTackToeLib /c scores.xsd).
The scores.cs is finally added to the project for
compilation.
-->
<scores>
      <entry player="Venkat" points="0" />
</scores>
```

The schema generated using xsd as discussed above in the comment section is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="scores" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
  <xs:element name="scores" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="entry">
          <xs:complexType>
            <xs:attribute name="player" type="xs:string" />
            <xs:attribute name="points" type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Finally, the class generated from this schema is shown below:

```
//------------------------------------------------------------------------
---------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.1.4322.573
//
//     Changes to this file may cause incorrect behavior and will be
lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------
---------

//
// This source code was auto-generated by xsd, Version=1.1.4322.573.
//
```

```
namespace TickTackToeLib {
    using System.Xml.Serialization;


    /// <remarks/>
    [System.Xml.Serialization.XmlRootAttribute(Namespace="",
IsNullable=false)]
    public class scores {

        /// <remarks/>
        [System.Xml.Serialization.XmlElementAttribute("entry",
Form=System.Xml.Schema.XmlSchemaForm.Unqualified)]
        public scoresEntry[] Items;
    }

    /// <remarks/>
    public class scoresEntry {

        /// <remarks/>
        [System.Xml.Serialization.XmlAttributeAttribute()]
        public string player;

        /// <remarks/>
        [System.Xml.Serialization.XmlAttributeAttribute()]
        public string points;
    }
}
```

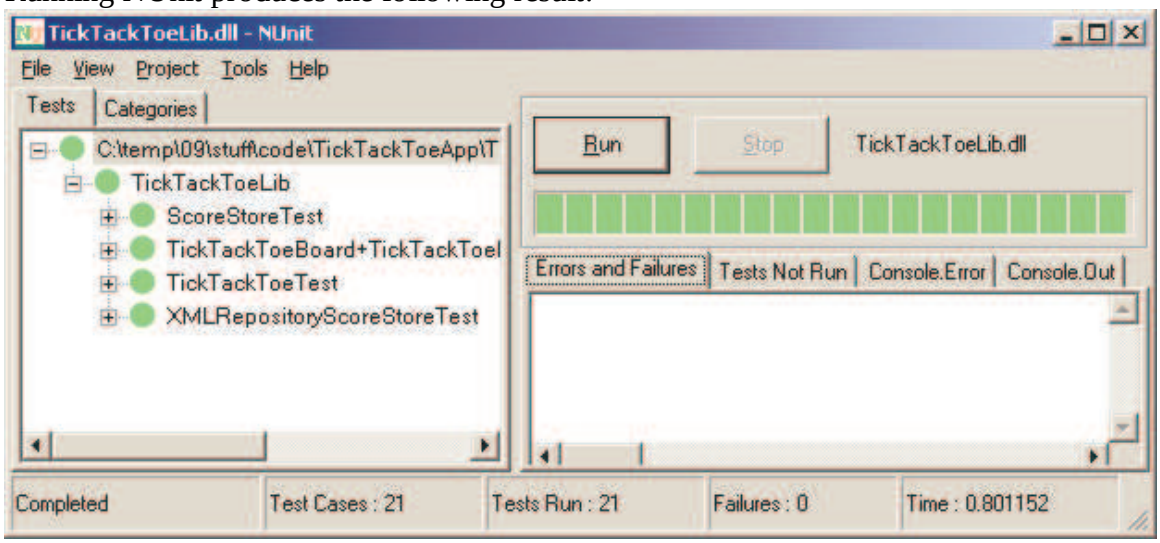
Now, we will include this scores.cs file into our project and use it in our XMLRepository as shown below:

```
using System;
using System.IO;
using System.Xml.Serialization;
using System.Collections;

namespace TickTackToeLib
{
      public class XMLRepository : IRepository
      {
            private string xmlFileName;
            private scores theScores = new scores();

            public XMLRepository(string theRepositoryFile)
            {
                  xmlFileName = theRepositoryFile;
                  if (File.Exists(xmlFileName))
                  {
                        XmlSerializer serializer =
                              new XmlSerializer(typeof(scores));
                        FileStream stream = new FileStream(xmlFileName,
                              FileMode.Open, FileAccess.Read);
                        theScores =
                              serializer.Deserialize(stream) as scores;
                        stream.Close();
                  }
```

```
        else
        {
                theScores = new scores();
        }
}

#region IRepository Members

public int GetScore(string player)
{
        int result = 0;
        if (theScores.Items != null)
        {
                foreach(scoresEntry entry in theScores.Items)
                {
                        if(entry.player == player)
                        {
                                result
                                        = Convert.ToInt32(
                                        entry.points);
                        }
                }
        }
        return result;
}

public void SetScore(string player, int score)
{
        bool found = false;

        if (theScores.Items != null)
        {
                foreach(scoresEntry entry in theScores.Items)
                {
                        if(entry.player == player)
                        {
                                entry.points = score.ToString();
                                found = true;
                        }
                }
        }

        if (!found)
        {
                ArrayList scoresAsArrayList = new ArrayList();
                if (theScores.Items != null)
                {
                        scoresAsArrayList.AddRange(
                                theScores.Items);
                }
                scoresEntry newEntry = new scoresEntry();
                newEntry.player = player;
                newEntry.points = score.ToString();
                scoresAsArrayList.Add(newEntry);
                theScores.Items = (scoresEntry[])
                        scoresAsArrayList.ToArray(
                        typeof(scoresEntry));
```

```
                }

                // update the scores file
                XmlSerializer serializer
                        = new XmlSerializer(typeof(scores));
                FileStream stream = new FileStream(xmlFileName,
                        FileMode.Create, FileAccess.Write);
                serializer.Serialize(stream, theScores);
                stream.Close();
        }

        public System.Collections.Hashtable GetScores()
        {
                Hashtable scores = new Hashtable();

                if (theScores.Items != null)
                {
                        foreach(scoresEntry entry in theScores.Items)
                        {
                                scores.Add(entry.player, entry.points);
                        }
                }

                return scores;
        }

        #endregion
    }
}
```

Running NUnit produces the following result:



Let make the following change to the windows form to use the XMLRepository:

We change

```
        private ScoreStore theStore
                = new ScoreStore(new InMemoryRepository());
```

to

```
            private ScoreStore theStore
                  = new ScoreStore(new XMLRepository("scores.xml"));
```

Playing the game a few times may create an XML store like this one:

```
<?xml version="1.0"?>
<scores xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <entry player="Venkat" points="1" />
  <entry player="Jimmy" points="1" />
</scores>
```

Benefits of Mock Object

We can see the benefit that our Mock object gave us from the above example. The InMemoryRepository served as the Mock until we created a real storage (and still serves as a mock for continued testing and improvements). The benefits are:

- Instead of getting into the complexity of storage, we were able to focus on simply the functionality that our application needs. We were able to understand what needs to be done, do it without spending much time and effort on persistence.
- The functionality of our application can be tested and it can be improved without depending on any particular storage. This isolation from actual repository helps us great deal in working with the application.
- When developing an application, you can easily separate the details of complexity from what you really depend upon, which is interface that will some how be implemented by yourself or someone else later on.
- When some thing fails, you can easily identify the layer from which the problem manifests as error or bug.
- If we are dependent on a third party product, our mock can simulate various failure conditions and help us write a robust code that will be resilient to changes and failures in the third party code.
- The IOC or DIP used in creating the Mock has one significant advantage. We can easily switch the repository now to use a real database, or a web service or just about any thing else we want to do. All we need is to write an adapter that supports the IRepository interface and communicates with our storage mechanism.

How to create a Mock?

One way to create a mock is to write one ourselves. Alternately you may use a framework. There are a handful of frameworks for this in Java: Mockrunner, DynaMock, JMock, EasyMock, etc. Since are using .NET here, I will show you an example of using the Easy Mock.

Easy Mock

The Easy Mock framework[7] creates a mock object on the fly for you! You simply go to a controller and "tell" it that you want a mock that implements a certain interface. A mock is created for you at that moment at run time. The Mock works in two modes. A mode where it listens or learns and then a play back mode where it simply responds the way you asked it to. During the playback mode, you can ask him to verify if (a) all methods you expected to be called were indeed called and (b) if they were called in the order in which you wanted them to be called.

Using Easy Mock

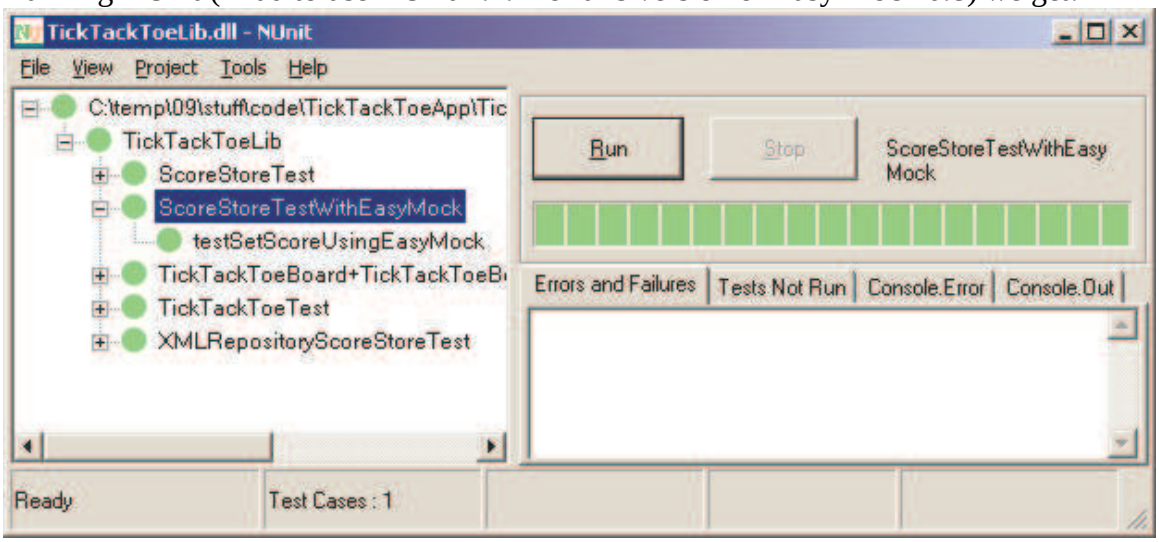
Let's just create another test to use the Mock objects for the repository. This mock will implement our IRepository interface.

```
using System;
using NUnit.Framework;
using EasyMockNET;

namespace TickTackToeLib
{
	[TestFixture]
	public class ScoreStoreTestWithEasyMock
	{
		[Test]
		public void testSetScoreUsingEasyMock()
		{
			//*** Note, I had to use NUnit2.1.4 with this
			// Version of EasyMock assembly -
			// EasyMockNET.NUnit.fw1.1.dll

			string PLAYER = "Venkat";

			// Prepare the Mock
			IRepository theRepository;

			IMockControl theMockControl =
				EasyMock.ControlFor(typeof(IRepository));

			theRepository
				= (IRepository) theMockControl.GetMock();

			theRepository.GetScore(PLAYER);
			theMockControl.SetReturnValue(0, 2);
				// We just *told* the Mock that if
				// GetScore is called with PLAYER,
				// it should return a 0. We also
				// want it to do that twice.

			theRepository.SetScore(PLAYER, 1);
			theMockControl.SetVoidCallable();
				// Nothing for the Mock to return
				// for this one

			theRepository.GetScore(PLAYER);
			theMockControl.SetReturnValue(1);
```

```
                    // Ask Mock to Play
                    theMockControl.Activate();

                    ScoreStore theScoreStore
                          = new ScoreStore(theRepository);
                    int score = theScoreStore.GetScore(PLAYER);
                    theScoreStore.UpdateScore(PLAYER);
                    Assert.AreEqual(score + 1,
                          theScoreStore.GetScore(PLAYER));

                    // Ask Mock how things went
                    theMockControl.Verify();
              }
        }
}
```

We first create a Mock controller for the IRepository interface. This controller creates a Mock object for us. Then we instruct the mock to receive certain messages like GetScore and what response it should give. Finally, we put it in play back mode by calling Activate. When the test is complete, we ask the mock controller to verify all calls were made and in order.

Running NUnit (I had to use NUnit 2.1.4 for this version of Easy Mock 0.8) we get:



Conclusion

In Part I we discussed test first coding. In this part (Part II) we saw how Mocks provide an easy and effective way to isolate your code from its complex dependencies. It makes it easier to develop systems, to make them more testable and also to layer it for easy replacement with alternative implementations. You may hand grow a Mock yourself or use frameworks that allow you to create the Mock. In the next article (Part III) we will discuss continuous integration.

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.
7. http://sourceforge.net/projects/easymocknet/