http://www.primaryobjects.com/CMS/Article108

SourceCode working:
https://github.com/techasuran/Asp.Net_DesignPatternsApps/tree/master/RepositoryPatternTest/RepositoryPatternTest

Introduction

Writing a C# ASP .NET web application which utilizes a database can create increasingly complex code. The more complex code gets, the more difficult it becomes to debug, maintain, and enhance. The Repository design pattern is a way of introducing architecture into your C# ASP .NET web application, which creates a clear layer of separation between your web forms and database accessible layers. The Repository pattern helps organize a web application to form a 3-tier architecture and provide loosly coupled classes, which can be reused in future C# ASP .NET web applications and easily updated.

In this article we'll implement a Repository Pattern for an example C# ASP .NET application which deals with planets in the solar system. The example application will contain 3-tiers, including the user interface, business logic layer, and database layer via the Repository design pattern.

C# .NET Repository Design Pattern   A Repository Pattern is not THE Repository Pattern

artin Fowler has described the traditional definition of the Repository design pattern as, "Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects". This traditional definition describes the Repository design pattern as a class which closely maps to a database table, containing Insert, Update, Select, Delete, and collection functions for persisting a particular class object in the database. While this describes the traditional usage of the Repository pattern, many developers create their own summarized versions of the pattern, which are still equally effective. The main goal of the Repository design pattern is to decouple the 3-tier architecture of a C# ASP .NET application. In our example, we'll be creating our own implementation of the Repository Pattern. While our version may not contain the full set of CRUD functions, our implementation will still benefit from the Repository pattern's features of decoupled code and logical separation of layers.

Accessing the Database Old School Style

Many developers are familiar with accessing a database using code similar to the following in C# ASP .NET:

```
static void Main(string[] args)
{
   SqlConnection MyConnection = new
SqlConnection("YourConnectionString");

try
{
   MyConnection.Open();

   SqlCommand MyCommand = new SqlCommand("SELECT * FROM Planet",
MyConnection);
   SqlDataReader MyReader = MyCommand.ExecuteReader();
   while (MyReader.Read())
   {
      // Read from the database and do things.
      Planet planet = new
Planet(Convert.ToInt32(MyReader["ID"].ToString()),
MyReader["Name"].ToString(),
Convert.ToInt32(MyReader["IsTerraformable"].ToString()));

      Console.WriteLine("Planet " + planet.Name);
   }

   MyReader.Close();
}
catch (Exception excep)
{
}
finally
{
   if (MyConnection != null && MyConnection.State ==
ConnectionState.Open)
   {
      MyConnection.Close();
   }
}
}
```

Notice that the above code is quite straight-forward. We simply open a database connection, read a set of planets from the database table, and perform processing with the data. All of the code for accessing the database, performing any business logic, and displaying to the user interface is located in one big main function. While this code is easy to read, since everything is located in one place, this code may suffer from maintaince issues as the code expands over time. Consider what would happen if the user interface was much more complex that `simply` writing to the Console window. The code could drastically expand just to display the information to the user. To resolve this potential issue, the code can be pulled apart, separated in to layers, and re-organized using the Repository design pattern to provide a much more maintainable C# ASP .NET application.

A Cleaned Up Way of Accessing the Database with 3-Tiers

You can download the complete source code for this project by clicking here.

Using the Repository design pattern, we can actually completely separate the layers of concerns in the above example code, for accessing the planets in the solar system via the database. The code below shows an example of a new main method where the code for displaying to the user interface, processing business logic, and accessing the database are separated:

```
static void Main(string[] args)
{
   // Load planets from the MSSQL repository.
   PlanetBusiness planetBusiness = new PlanetBusiness(new
MSSQLPlanetRepository());

   List<Planet> planetList = planetBusiness.GetPlanets();
   foreach (Planet planet in planetList)
   {
      Console.WriteLine(planet.PlanetId + ". " + planet.Name);
   }
}
```

Notice in the above code, there are no database accessible functions. We've also taken out any business logic processing that might occur after reading from the database. The only code included in our main method now is code which displays to the user interface. This clearly makes the job of maintaining the web application code much easier. Let's take a look at what's behind the scenes in the above example, which puts the pieces together to implement the Repository pattern.

We Can't Start Without a Type

Before we actually begin with the Repository pattern, we need to define a basic type library. In this library, we'll only have a single type called "Planet". This is a basic C# .NET class which has the required fields for working with a planet, most likely the same fields as the database table.

It's important to note that by creating your own light-weight type library, instead of using a potentially heavier ORM-generated one, you can be sure that your C# ASP .NET web application remains decoupled and completely separated in layers. This allows you the potential of swapping out different repositories at any time in the future, without having to change code in your main method. Since the main method deals with the Planet type, as long as the type doesn't change, the different Repositories can populate the Planet type as they wish and the main method will never know the difference.

```
public class Planet
{
   public int PlanetId;
   public string Name;
   public bool IsTerraform;

   public Planet()
   {
   }

   public Planet(int planetId, string name, bool isTerraform)
   {
      PlanetId = planetId;
      Name = name;
      IsTerraform = isTerraform;
   }
}
```

The above is a simple definition of the Planet type. It appears similar to the database table Planet, which contains an ID, Name, and boolean flag for Terraform. Any repository that we implement will populate the Planet type with data in their own way. Our main method, and other areas of our C# ASP .NET web application never need to know the details about how the repository actually populates the Planet type; they only concern themselves with using the type itself.

The Repository Pattern Starts With a Lonely Interface

As with most design patterns, the Repository pattern begins with a single interface, which outlines the methods that the data repository class will be able to perform. An example Repository interface is shown below:

```
public interface IPlanetRepository
{
   List<Planet> GetPlanets();
   List<Planet> GetStars();
}
```

In the above interface, we've simply defined two methods. One method fetches the planets in the solar system, while the other method fetches stars. One could easily expand this Repository pattern interface to include a full set of data access functions for working with a planet, such as Insert, Delete, Update, etc. For this example, we'll stick with these two basic functions for loading planets. The next step is to create a concrete repository class, which implements the interface.

An SQL Server Repository is Easy

Creating our first concrete repository is fairly simple. We'll simply implement the Repository pattern interface and write the required code for accessing the MSSQL database and populating the Planet types.

```
public class MSSQLPlanetRepository : IPlanetRepository
{
#region IPlanetRepository Members

public List<Planet> GetPlanets()
{
   // Simulate returning the list of data from the database.
   List<Planet> planetList = new List<Planet>();

   // SqlConnection code would actually go here, this is to keep
things simple.
   planetList.Add(new Planet(1, "Mercury", false));
   planetList.Add(new Planet(2, "Venus", true));
   planetList.Add(new Planet(3, "Earth", true));
   planetList.Add(new Planet(4, "Mars", true));
   planetList.Add(new Planet(5, "Jupiter", false));
   planetList.Add(new Planet(6, "Saturn", false));
   planetList.Add(new Planet(7, "Uranus", false));
   planetList.Add(new Planet(8, "Neptune", false));
   planetList.Add(new Planet(9, "Pluto", false));

   return planetList;
}

public List<Planet> GetStars()
{
   // Simulate returning the list of data from the database.
   List<Planet> planetList = new List<Planet>();

   // SqlConnection code would actually go here, this is to keep
things simple.
   planetList.Add(new Planet(1, "Sun", false));

   return planetList;
}
#endregion
}
```

The first item to note is that in this example, we're not actually reading from an MSSQL database. To keep the code simple, we're simply populating a list of Planets in the code. It is safe to assume that in place of the code adding planets to the array, would be code which loops through a DataReader and creates a planet for each read.

By returning test data from this repository, rather than reading from the actual database, we're actually showing off another advantage of the Repository design pattern. The advantage is the ability to easily create unit-testable methods in our applications, which return sample data without hitting the actual database. It's important to have consistent data in unit tests, whereas data from a live database may change over time. The Repository pattern allows us to easily create and swap in/out a unit test repository to verify business logic remains in-tact.

With the SQL Server repository implemented, we can now create our business logic layer.

The Business Layer Glues it Together

The simple .NET application we're creating in this article actually has minimal use for a business logic layer. However, in more complex C# ASP .NET applications, a business logic layer can greatly enhance the maintainability of a project. To optimally implement the Repository pattern, we'll need to create our business logic layer.

The business layer sits between the user interface and the Repository pattern data class. The business layer can be considered as a filter for the Repository pattern, calling the repository to obtain data and passing the processed results to the calling class (such as the main method in our user interface).

```
public class PlanetBusiness
{
   IPlanetRepository _repository;

   public PlanetBusiness(IPlanetRepository repository)
   {
      _repository = repository;
   }

   public List<Planet> GetPlanets()
   {
      return _repository.GetPlanets();
   }

   public List<Planet> GetStars()
   {
        return _repository.GetStars();
    }

    public List<Planet> GetTerraformPlanets()
    {
        // Return the list of filtered data from the database.
        List<Planet> planetList = _repository.GetPlanets();

        var results = from p in planetList
            where p.IsTerraform == true
            select p;

        return results.ToList();
    }
}
```

Notice in the above code, we've defined our business logic class to include a private instance of a planet Repository class. This allows our business class to implement any type of concrete planet repository class. For example, you could create PlanetBusiness to implement the MSSQL repository or you could create the class to implement a completely different repository, such as a test repository, Oracle, MySQL, etc. The important piece to note is that the member variable is a generic interface, which can implement any type of concrete planet repository.

The remainer of PlanetBusiness models the repository interface, with regard to the functions the calling class may want to use. We create methods for GetPlanets(), GetStarts(), and a new function called GetTerraformPlanets(), which actually calls the repository's method for GetPlanets(), but performs additional filtering of the data to return only those planets which can be terraformed.

Putting it All Together

With the layers defined, we can put the Repository pattern in action in our main method by creating an instance of the BusinessPlanet business logic layer, and passing in our desired instance of the concrete repository class.

```
static void Main(string[] args)
{
    // Load planets from the MSSQL repository.
    PlanetBusiness planetBusiness = new PlanetBusiness(new
MSSQLPlanetRepository());

    List<Planet> planetList = planetBusiness.GetPlanets();
    foreach (Planet planet in planetList)
    {
        Console.WriteLine(planet.PlanetId + ". " + planet.Name);
    }
}
```

You can see in the above code how we've implemented the PlanetBusiness class with a concrete instance of the MSSQLPlanetRepository. When we call the generic business function GetPlants(), the business layer calls the MSSQL repository for the planet data. What if we wanted to move the database to Oracle? Would this effect the whole program?

Another Repository, Minimal Code

We can actually create another planet repository quite easily, effectively allowing us to change data sources, without touching any other areas of the application. This is an extremely powerful feature and the main reason to use the repository pattern. By limiting the areas of the web application that need code changes, we can make major alterations to the application with minimal chance for errors.

```
public class OracleRepository : IPlanetRepository
{
   #region IPlanetRepository Members

   public List<Planet> GetPlanets()
   {
      // Simulate returning the list of data from the database.
      List<Planet> planetList = new List<Planet>();

      // OracleConnection code would actually go here, this is to keep
things simple.
      planetList.Add(new Planet(1, "Mercury", false));
      planetList.Add(new Planet(2, "Venus", true));
      planetList.Add(new Planet(3, "Earth", true));
      planetList.Add(new Planet(4, "Mars", true));
      planetList.Add(new Planet(5, "Jupiter", false));
      planetList.Add(new Planet(6, "Saturn", false));
      planetList.Add(new Planet(7, "Uranus", false));
      planetList.Add(new Planet(8, "Neptune", false));
      planetList.Add(new Planet(9, "Pluto", false));
      planetList.Add(new Planet(10, "Planet X", false));

      return planetList;
}

   public List<Planet> GetStars()
   {
      // Simulate returning the list of data from the database.
      List<Planet> planetList = new List<Planet>();
```

```
        // OracleConnection code would actually go here, this is to keep
things simple.
        planetList.Add(new Planet(1, "Sun", false));
        planetList.Add(new Planet(2, "Alpha Centauri", false));

        return planetList;
    }

    #endregion
}
```

The above code defines another planet repository, this time working with an Oracle database. Again, the data loaded is hard-coded for simplicity, but would actually be loading from a database. To show differences in the data, the hard-coded values are slightly different than the MSSQL repository.

Calling Another Repository is the Same

To call our newly created Oracle planet repository, we simply implement the new repository in our business class, as follows:

```
static void Main(string[] args)
{
    // Load planets from the Oracle repository.
    PlanetBusiness planetBusiness = new PlanetBusiness(new
OraclePlanetRepository());

    List<Planet> planetList = planetBusiness.GetPlanets();
    foreach (Planet planet in planetList)
    {
        Console.WriteLine(planet.PlanetId + ". " + planet.Name);
    }
}
```

With this one small change, we've instantly changed the functionality of the C# ASP .NET web application from working with planets in an SQL Server database to working with an Oracle database. Further, for unit testing, we could even create a unit-testable repository with no database access at all.

Unit Testing is Boring, But at Least the Repository Makes it Easy

When performing unit testing against a live database, values can often change, making it difficult to verify that a variable holds a specific value. To resolve this type of unit-testing issue, we can create a special unit test Repository, which contains static data or hard-coded values.

The data returned would then be processed by the business layer and can be unit tested accordingly.

```
public class TestRepository : IPlanetRepository
{
#region IPlanetRepository Members

public List<Planet> GetPlanets()
{
// Simulate returning the list of data from the database.
List<Planet> planetList = new List<Planet>();

planetList.Add(new Planet(1, "Earth", true));
planetList.Add(new Planet(2, "Test Planet", true));

return planetList;
}

public List<Planet> GetStars()
{
// Simulate returning the list of data from the database.
List<Planet> planetList = new List<Planet>();

planetList.Add(new Planet(1, "None", false));

return planetList;
}

#endregion
}
```

In the above code we've defined yet another planet repository, which loads a set of hard-coded data. This repository makes it easy to unit test and verify proper functioning of the business logic and layers.

Combining Repositories For Some Action

To really see the power behind the repository pattern, lets implement all three repository types in the main method and see how the different data loads:

```
static void Main(string[] args)
{
   // Load planets from the MSSQL repository.
   PlanetBusiness planetBusiness = new PlanetBusiness(new
MSSQLPlanetRepository());
```

```
    TestPlanets(planetBusiness);

    // Load planets from the Oracle repository.
    planetBusiness = new PlanetBusiness(new OracleRepository());
    TestPlanets(planetBusiness);

    // Load planets from the Test repository.
    planetBusiness = new PlanetBusiness(new TestRepository());
    TestPlanets(planetBusiness);

    Console.ReadKey();
}

private static void TestPlanets(PlanetBusiness planetBusiness)
{
// Basic driver class to test our planet business class and display
output.
Console.WriteLine();
Console.WriteLine("Planets:");
Console.WriteLine();

List<Planet> planetList = planetBusiness.GetPlanets();
foreach (Planet planet in planetList)
{
Console.WriteLine(planet.PlanetId + ". " + planet.Name);
}

Console.WriteLine();
Console.WriteLine("Terraformable Planets:");
Console.WriteLine();

planetList = planetBusiness.GetTerraformPlanets();
foreach (Planet planet in planetList)
{
Console.WriteLine(planet.PlanetId + ". " + planet.Name);
}
}
```

The above code has combined the different planet repositories into a single main method. The data displayed to the user interface via the TestPlanets() method will change depending on which concrete repository is implemented in the PlanetBusiness class.

We can actually take this one step further with .NET reflection to allow us to implement a concrete repository at run-time without re-compiling code.

Using Reflection to Super-Charge your Repository Pattern

Until now, the previous examples showed static pre-compile time implementations of the repository pattern. We implemented an MSSQL, Oracle, and Test planet repository. However, each time we changed the repository we needed to re-compile and run the application. In a live enviornment, this may not be optimal, especially when automatic unit testing scripts may be running. In this case, we can use C# .NET reflection to automatically implement the desired concrete repository at run-time, by reading the web.config.

To begin, simply add the following code to your web.config file, which defines the concrete repository class and assembly file to load the repository from.

```
<appSettings>
<!-- The assembly name of the default repository. Use the format
Fully.Qualified.Class.Name, Assembly.FileName -->
<add key="DefaultRepository"
value="RepositoryPatternTest.Repository.Repositories.MSSQLPlanetReposi
tory, RepositoryPatternTest.Repository" />
</appSettings>
```

Then modify the main method to implement this concrete repository in the business layer, as follows:

```
static void Main(string[] args)
{
   // Load planets from the default repository as specified in the
web.config. Allows for changing the repository at run-time, without
changing code!
   Type obj =
Type.GetType(ConfigurationManager.AppSettings["DefaultRepository"]);
   ConstructorInfo constructor = obj.GetConstructor(new Type[] { });
   IPlanetRepository defaultRepository =
(IPlanetRepository)constructor.Invoke(null);

   PlanetBusiness planetBusiness = new
PlanetBusiness(defaultRepository);
   TestPlanets(planetBusiness);

   Console.ReadKey();
}
```

In the above code, we're first obtaining the type for the repository by reading the web.config and locating the class. We then invoke the class's constructor to create an instance of the concrete repository. This is the same is calling: new MSSQLPlanetRepository(). We can then assign the concrete repository to the PlanetBusiness business logic layer as in the previous examples.

You can see that by adding reflection into the process, you could easily expand the capability of your architecture to support loading multiple repositories at run-time for different enviornments, such as automatic unit testing, QA server testing, and production scenerios.

Conclusion

The Repository design pattern can help to cleanup and separate the layers of a C# ASP .NET web application. The pattern helps provide a decoupled 3-tier architecture, which improves the maintainability of an application and helps reduce errors. By enhancing the repository pattern with features such as .NET reflection, you can implement different concrete repositories at run-time, helping to more easily support automatic unit testing enviornments, QA, and production development.

About the Author

This article was written by Kory Becker, founder and chief developer of Primary Objects, a software and web application development company. You can contact Primary Objects regarding your software development needs at http://www.primaryobjects.com