Enterprise Design Patterns

Design Patterns - MVP - Model View Presenter Pattern-Part 1

Introduction:

As we all know many design patterns and using them in implementing business components and services in our applications but still we will be facing issues/challenges with our applications. Also day by day business needs and priorities will be changing. If we closely observe at number of issues, defects and challenges we are facing is in UI and presentation layers. Though some defects are related to business logic and business rules we may need to fix them in UI and presentation layers as we might tightly integrated business logic in UI and presentation tiers. The reason behind this is we haven't focused on implementing right design patters in our applications. Let us go through step by step and understand how to implement and use presentation pattern in our applications.

Problem Statement:

1. Already using different patterns in application but still maintaining application is difficult.
2. Using VS Test, NUnit, MBUnit etc to test business logic layer, but still some defects are exists in the application as business logic involved in presentation layer.
3. Used Presentation Layer, Business Logic Layer, Data Access Layer in the application but still some times need to write redundant code in presentation layer to consume or call other modules or other use cases.
4. Integration defects are getting injected when we make some changes in integrated modules.
5. Defect fixing and enhancements are taking more time to analyze the presentation tier logic and its integration dependencies and causing for opening new defects.
6. ASP.NET MVC cannot be chosen as UI is complex to build.

Root Cause of the Problem:

In Presentation layer,
1. A page or form contains controls that display application domain data.
   A user can modify the data and submit the changes.
   The page retrieves the domain data, handles user events, alters other controls on the page in response to the events, and submits the changed domain data.
   Including the code that performs these functions in the Web page makes the class complex, difficult to maintain, and hard to test.
2. In addition, it is difficult to share code between Web pages that require the same behavior.
3. UI Layer, UI Logic, Presentation Logic, Business Logic are tightly coupled.
4. Presentation layer is taking care of integrating modules or use cases.

Solution:

1. Chose a best Presentation Layer Pattern to separate the UI Layer, UI Logic and Presentation Logic and Business Logic as separate layers to make the code easier to understand and maintain.
2. Enable loose coupling while developing modules or any use cases.
3. Maximize the code that can be tested with automation. (Views are hard to test.)
4. Share code between pages that require the same behavior.
5. Separate the responsibilities for the visual display and the event handling behavior into different classes named, respectively, the view and the presenter or controller or ViewModel.

Benefits of using Presentation Pattern:

1. Modularity
2. Test driven approach - maximize the code that can be tested with automation
3. Separation of concerns
4. Code sharing between pages and forms
5. Easy to maintain

What are the presentation layer patterns available?

MVC (Model View Controller)
MVP (Model View Presenter) or (Model Passive View, Supervisor Controller)
MVVM (Model View ViewModel)

MVC vs MVP vs MVVM:

1. Model and View represents same in all the above 3 patterns?
Yes
2. Controller, Presenter, and ViewModel purpose is same in all the above 3 patterns?
Yes
3. Communication and flow of Model, View with Controller, Presenter, and ViewModel is same?
No, that is the reason these 3 patterns exists.
4. Are these patterns replacement of PL (Presentation Layer), BLL (Business Logic Layer) and DAL (Data Access Layer)
No, these patterns are for separating the UI and UI Logic from Presentation Logic
and enables the loose coupling.

Choose the best Presentation Layer Pattern:

MVP

1. Binding via a datacontext is not possible

1. Binding via a datacontext is not possible
2. Complex UI Design
3. Best for Windows Forms, ASP.NET Web Forms & Sharepoint Applications

MVC

1. Best for ASP.NET with Simple UI
2. Disconnected Model (View separation from all other layers)

Note: Here I am not focusing on MVC VM (MVC ViewModel from MVC3) and ASP.NET MVVM with Dependency Injection.
Read more about ASP.NET MVVM http://aspnetmvvm.codeplex.com/
MVC ViewModel http://msdn.microsoft.com/en-us/vs2010trainingcourse_aspnetmvc3fundamentals_topic7.aspx

MVVM

1. Binding via a datacontext is possible
2. Connected Model
3. Best for WPF and Silverlight applications

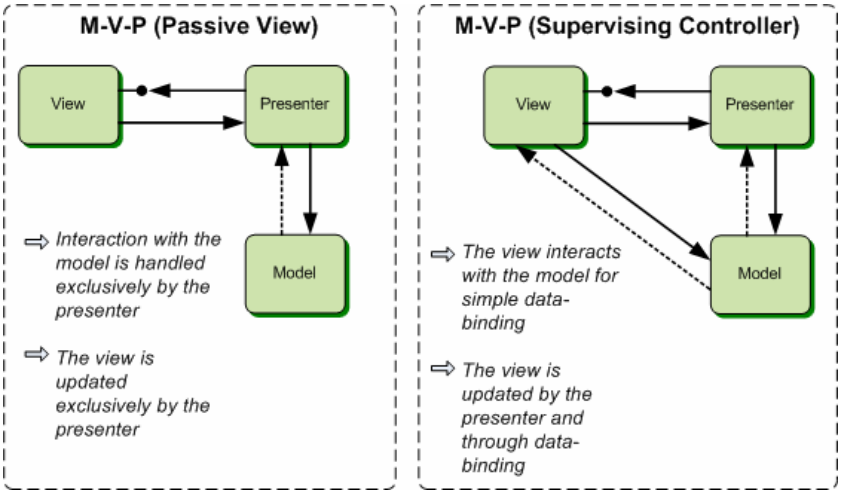
Read more about MVVM at http://code.msdn.microsoft.com/Design-Patterns-MVVM-Model-d4b512f0

How to Implement MVP?

1. The View class (the Web page or user control) manages the controls on the page and it forwards user events to a presenter class and expose properties that allow the presenter to manipulate the view's state.

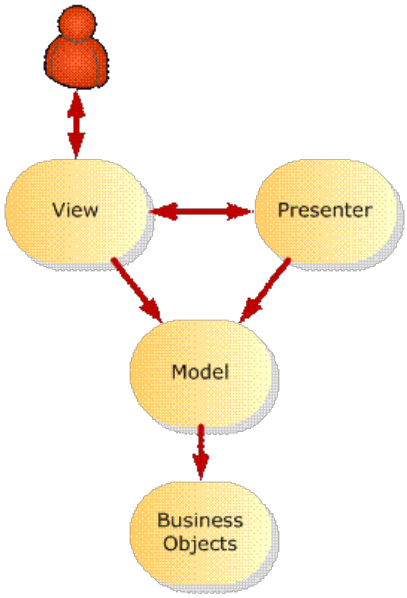
2. The Presenter contains the logic to handle and respond to the events, update the model (business logic and data of the application) and, in turn, manipulate the state of the view.

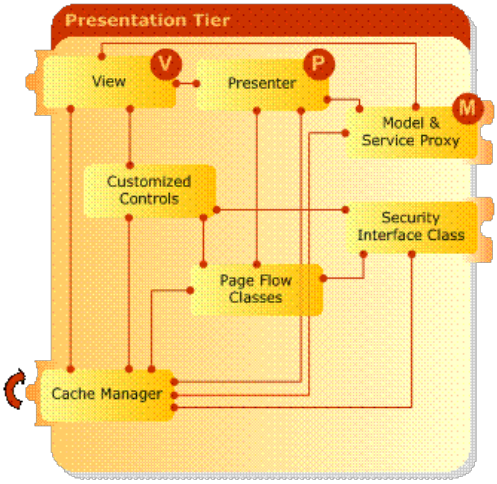
3. The View Interface, to facilitate testing the presenter, the presenter will have a reference to the view interface instead of to the concrete implementation of the view. View interface expose the view's state elements. By doing this, we can easily replace the real view with a mock implementation to run tests.



One of my client's application architecture:

I will explain soon, how we implented and what components we have used.





Implementing a View Class:

The view (a web page, a user control, or a master page) contains user interface elements.
The view should forward user events to the presenter and expose properties or methods for the presenter to manipulate the view's state.

How View Updates?

When the model is updated, the view also has to be updated to reflect the changes. View updates can be handled in several ways. The Model-View-Presenter variants, Passive View and Supervising Controller, specify different approaches to implementing view updates.

In Passive View, the interaction with the model is handled exclusively by the presenter; the view is not aware of changes in the model. The presenter updates the view to reflect changes in the model.

In Supervising Controller, the view interacts directly with the model to perform simple data-binding that can be defined declaratively, without presenter intervention.
The presenter updates the model; it manipulates the state of the view only in cases where complex UI logic that cannot be specified declaratively is required.

Examples of complex UI logic might include changing the color of a control or dynamically hiding/showing controls.

When to use Passive View or Supervising Controller

If testability is a primary concern in our application, Passive View might be more suitable because we can test all the UI logic by testing the presenter.

If we prefer code simplicity over full testability, Supervising Controller might be a better option because, for simple UI changes, we do not have to include code in the presenter that updates the view.

Passive View usually provides a larger testing surface than Supervising Controller because all the view update logic is placed in the presenter.
Supervising Controller typically requires less code than Passive View because the presenter does not perform simple view updates.

If we are using the Supervising Controller variant, the view should also perform direct data binding to the model (for example, using the ASP.NET built-in ObjectDataSource control).
In cases where the presenter exclusively handles the interaction with the model, the ObjectContainerDataSource Control will help in implementing data binding through the presenter.
The ObjectContainerDataSource control was designed to facilitate data binding in a Model-View-Presenter scenario, where the view does not have direct interaction with the model.

View Interaction with the Presenter

There are several ways that the view can forward user gestures to the presenter.
1. The view directly invokes the presenter's methods.
This approach requires, implement additional methods in the presenter and couples the view with a particular presenter.
2. Having the view raise events when user actions occur.
This approach requires code in the view to raise events and code in the presenter to subscribe to the view events.

The benefit to the second approach is that there is less coupling between the view and the presenter than with the first approach.

View Interaction with the Model

We can implement the interaction with the model in several ways.
For e.g, we can implement the Observer pattern. I.e. the presenter receives events from the model and updates the view as required.
Another approach is to use an application controller to update the model.

Read more about Observer Pattern and its implementation here http://code.msdn.microsoft.com/Dive-into-Observer-Pattern-00fa8573

Implementing a Presenter Class:

The presenter should handle user events, update the model, and manipulate the state of the view.
Usually, the presenter is implemented using Test Driven Development (TDD).
Implementing the presenter first allows us to focus on the business logic and application functionality independently from the user interface implementation.
When implementing the presenter, we also need to create simple view and model objects to express the interaction between the view, the model, and the presenter.
In order to test the presenter in isolation, make the presenter reference the view interface instead of the view concrete implementation.

By doing this, we can easily replace the view with a mock implementation when writing and running tests.

Presenter Interaction with the View:

The communication with the view is usually accomplished by setting and querying properties in the view to set and get the view's state, respectively and invoking methods on the view.
For e.g,
1. A view could expose a Customer property that allows the presenter to set the customer that the view should display.
2. The presenter could invoke a method named ShowCustomer(Customer) that indicates to the view that it has to display the customer passed by parameter.

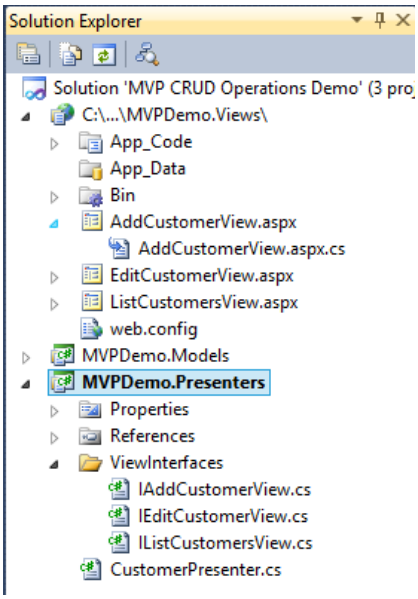
Implementing a View Interface:

The view interface should expose the view's state.
Typically, a view interface contains properties for the presenter to set and query the view's state.
Exposing properties over methods in the view usually keeps the presenter simpler because it does not need to know about view implementation details, such as when data is to be bound to user interface controls.
Depending on how the view interacts with the presenter, the view interface might also have additional elements.
If the view interacts with the presenter by raising events, the view interface will include event declarations.

Steps to Implement:

Let us go through the few code snippets from the attached sample.

Create a solution and add the projects with specified names as shown below.



Step 1: Create a simple Model - CustomerModel.cs

C#

```
namespace MVPDemo.Models
{
    public class CustomerModel
    {
        private string firstName;
        private string lastName;

        public string FirstName
        {
            get { return firstName; }
            set
            {
                firstName = value;
            }
        }

        public string LastName
        {
            get { return lastName; }
            set
            {
                lastName = value;
            }
        }

        public string FullName
        {
            get
```

```
            {
                return firstName + " " + lastName;
            }
        }

        public CustomerModel()
        {
        }

        public CustomerModel(string firstName, string lastName)
        {
            this.FirstName = firstName;
            this.LastName = lastName;
        }
    }
}
```

Step 2: Create a View Class UI (AddCustomer.aspx.cs)

HTML

```
<form id="form1" runat="server">
    <div>
        <asp:Label ID="lblMessage" runat="server" /><br />
        <br />
        <table>
            <tr>
                <td>
                    First Name:
                </td>
                <td>
                    <asp:TextBox ID="txtFirstName" runat="server" MaxLength="5" />
                    <asp:RequiredFieldValidator ID="RequiredFieldValidator3" ControlToValidate="txtFirstName"
                        runat="server" ErrorMessage="Customer First Name must be provided" />
                </td>
            </tr>
            <tr>
                <td>
                    Last Name:
                </td>
                <td>
                    <asp:TextBox ID="txtLastName" runat="server" MaxLength="40" />
                    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" ControlToValidate="txtLastName"
                        runat="server" ErrorMessage="Customer Last Name must be provided" />
                </td>
            </tr>
            <tr>
                <td align="center">
                    <asp:Button ID="btnAdd" runat="server" OnClick="btnAdd_OnClick" Text="Add Customer" />
                </td>
                <td>
                    <asp:Button ID="btnCancel" runat="server" OnClick="btnCancel_OnClick" Text="Cancel"
                        CausesValidation="false" />
                </td>
            </tr>
        </table>
    </div>
 </form>
```

Step 3: Create a View Interface (IAddCustomerView.cs)

C#

```
public interface IAddCustomerView
{
        string Message { set; }
        void AttachPresenter(CustomerPresenter presenter);

        /// <summary>
        /// No need to have a setter since we're only interested in getting the new
        /// <see cref="Customer" /> to be added.
        /// </summary>
        CustomerModel CustomerToAdd { get; }
}
```

Step 4: Create a Presenter (CustomerPresenter.cs)

C#

```
using System;
using System.Web;

using MVPDemo.Models;
using MVPDemo.Presenters.ViewInterfaces;
using System.Collections.Generic;

namespace MVPDemo.Presenters
{
    public class CustomerPresenter
    {
        private IAddCustomerView AddCustomerViewObject;

        public CustomerModel CustomerModelObject;

        public CustomerPresenter(IAddCustomerView CustomerViewObject, CustomerModel CustomerModelObject)
        {

            if (CustomerViewObject == null)
            {
                throw new ArgumentNullException("CustomerViewObject may not be null");
            }
            if (CustomerModelObject == null)
            {
                throw new ArgumentNullException("CustomerModelObject may not be null");
            }

            this.AddCustomerViewObject = CustomerViewObject;
            this.CustomerModelObject = CustomerModelObject;
        }

        public EventHandler CancelAddEvent;

        public EventHandler AddCustomerEvent;

        public EventHandler AddCustomerCompleteEvent;

        public void AddInitView()
        {
            AddCustomerViewObject.Message = "Use this form to add a new customer.";
        }

        public void AddCustomer()
        {
            AddCustomerEvent(this, null);
        }

        public void AddCustomer(bool isPageValid)
        {
            // Be sure to check isPageValid before anything else
            if (!isPageValid)
            {
                AddCustomerViewObject.Message = "There was a problem with your inputs.  Make sure you supplied everything and try again"
;
                return;
            }

            //Get the filled object from the Customer view.
            CustomerModelObject = AddCustomerViewObject.CustomerToAdd;
            HttpContext.Current.Session["CustomerModelObject"] = CustomerModelObject;


            // You could certainly pass in more than just null for the event args
            AddCustomerCompleteEvent(this, null);

            // By passing HTML tags from the presenter to the view, we've essentially bound the presenter to an HTML context.
            // You may want to consider alternatives to keep the presentation layer web/windows agnostic.
            AddCustomerViewObject.Message = "<span style=\"color:red\">The Customer added to database successfully.";
        }

        public void CancelAdd()
        {
            CancelAddEvent(this, null);
        }
    }
}
```

Step 5: Associate View Class with Presenter by using View Interface (AddCustomer.aspx.cs) we can create seperate presenters for each CRUD operation. Here I have created only presenter for all CRUD operations.

C#

```
private CustomerPresenter presenter;
```

```
public string Message
{
    set
    {
        lblMessage.Text = value;
    }
}

public void AttachPresenter(CustomerPresenter presenter)
{
    this.presenter = presenter;
}

public CustomerModel CustomerToAdd
{
    get
    {
        CustomerModel customer = new CustomerModel();
        customer.FirstName = txtFirstName.Text;
        customer.LastName = txtLastName.Text;
        return customer;
    }
}

protected void btnAdd_OnClick(object sender, EventArgs e)
{
    presenter.AddCustomer(Page.IsValid);
}

protected void btnCancel_OnClick(object sender, EventArgs e)
{
    presenter.CancelAdd();
}

//Base page overload method
protected override void PageLoad()
{
    // DaoFactory is inherited from BasePage
    CustomerModel CustomerModelObject = new CustomerModel();

    CustomerPresenter presenter = new CustomerPresenter(this, CustomerModelObject);
    this.AttachPresenter(presenter);

    presenter.AddCustomerCompleteEvent += new EventHandler(HandleAddCustomerCompleteEvent);
    presenter.CancelAddEvent += new EventHandler(HandleCancelAddEvent);

    presenter.AddInitView();
}

private void HandleAddCustomerCompleteEvent(object sender, EventArgs e)
{
    Response.Redirect("ListCustomersView.aspx?action=added");
}

private void HandleCancelAddEvent(object sender, EventArgs e)
{
    Response.Redirect("ListCustomersView.aspx");
}
```

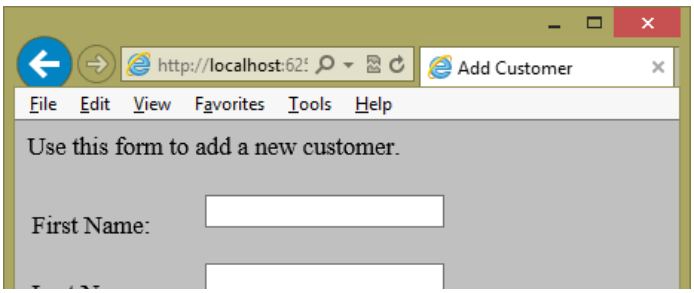
Additional Reference Links:

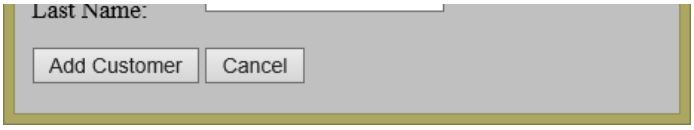
Composite Blocks with MVP http://webclientguidance.codeplex.com/

MVP with Web Forms http://webformsmvp.codeplex.com/

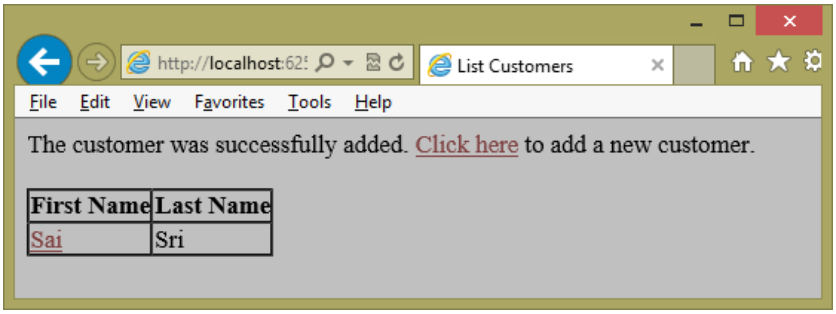
NuGet Framework for MVP http://www.nuget.org/packages/WebFormsMVP

Output: AddCustomer.aspx

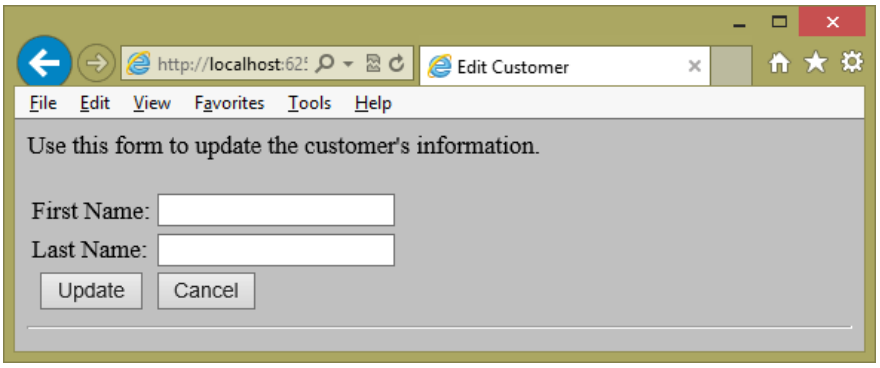




ListCustomer.aspx



EditCustomer.aspx



Few Disadvantage of MVP:

1. There are more solution elements to manage.
2. We need a way to create and connect views and presenters.
3. The model is not aware of the presenter. Therefore, if the model is changed by any component other than the presenter, the presenter must be notified. Typically, notification is implemented with events.

Thank you for reading my article. Drop all your questions/comments in QA tab give me your feedback with ★★★★★ star rating (1 Star - Very Poor, 5 Star - Very Good).

Why the Repository Pattern

By **Jamie Munro**, 30 May 2013

Introduction

The following article will explore a few different reasons that I believe in why the Repository pattern is an extremely useful paradigm that should be used more often - especially when it comes to ORM frameworks like Entity Framework.

After the incredible reaction to a recent blog post, Entity Framework Beginner's Guide Done Right, I feel like before writing some more code to further the basic example, I'll take a step back and explain my beliefs in the Repository pattern.

I was really overwhelmed with the reaction; what started with a simple 30 minute blogging effort has turned into something absolutely incredible. The original post was really a starting point about not placing direct querying and saving to the database mingled with the core code.

Please note, these are my thoughts on the pattern based on current and previous pain points that I've/am experiencing and I'd love to hear others input in making things better.

Before diving in, you can find the full source code up on GitHub: https://github.com/endyourif/RepoTest/

Making my code more readable

At its most basic level, the goal of a repository layer is to remove querying of content from my code. I personally find reading a function like this is much easier to read:

Collapse | Copy Code

```
// Display all Blogs from the database
var query = repo.GetAll();
Console.WriteLine("All blogs in the database:");
foreach (var item in query)
{
    Console.WriteLine(item.Name);
}
```

Than a function like this:

Collapse | Copy Code

```
// Display all Blogs from the database
var query = from b in db.Blogs
orderby b.Name
select b;
Console.WriteLine("All blogs in the database:");
foreach (var item in query)
{
    Console.WriteLine(item.Name);
}
```

In the first code example, I'm not boggled down by reading a LINQ query; I just need to deal with the results of the query from the function `GetAll`. This of course is probably a bad name; it might better to update the function name to be `GetAllBlogsOrderedByName`. It's of course very long, but incredibly descriptive. It leaves nothing to the imagination about what is happening in that function.

Segregating Responsibility

I work in a reasonably sized team. In this team, some people are great at writing highly optimized LINQ or SQL queries; while others on the team are great at executing the business logic pieces; while others are just juniors and need to earn my trust!

By adding a repository layer, I can segregate that responsibility. You're great at writing LINQ – perfect – you own the repository layer. If someone requires data access, they go through you. This will help prevent bad queries, multiple functions that achieve similar results (because somebody didn't know about XYZ function), etc.

Layering my code

I love the MVC design pattern – Model View Controller. Where the Controller talks to the Models to get some data and passes this through to the View for output.

So many times I have seen people mashing everything into the Controller – because it's the middle man. I actually wrote a blog post in 2009 that at the time was directed towards CakePHP: Keeping your CakePHP Controllers Clean. This of course applies to more than just CakePHP and even more than just controllers.

I like to start with a nice clean function that splits off into multiple small functions that perform the necessary work for me. E.g. error checking, business logic, and data access. When I have to debug this code, it makes it extremely easy to pin point the potential lines of code causing the issue; typically inside a single 10 line function; opposed to inside a large 100+ line function.

Adding a caching layer

This is not always an obvious feature, the average website or application might not require caching. But in the world I deal with daily, it's a must. By creating a repo layer, this is really simple.

Let's look at the previously discussed function `GetAllBlogsOrderedByName` with caching implemented:

Collapse | Copy Code

```
private static IOrderedQueryable<Blog> _blogs;
public IOrderedQueryable<Blog> GetAllBlogsOrderedByName()
{
    return _blogs ?? (_blogs = from b in _bloggingContext.Blogs
        orderby b.Name
        select b);
}
```

Knowing that my blogs only change when I save something, I can leverage C#'s ability of a static variable being in memory for the lifetime of the application running. If the `_blogs` variable is `null` I then and only then need to query for it.

To ensure this variable gets reset when I change something, I can easily update either my `AddBlog` or `SaveChanges` function to clear the results of that variable:

Collapse | Copy Code

```
public int SaveChanges()
{
    int changes = _bloggingContext.SaveChanges();
    if (changes > 0)
    {
        _blogs = null;
    }
    return changes;
}
```

If I was really courageous, I could even re-populate the `_blogs` variable instead of simply clearing it.

This example of course is only a starting point where I'm leveraging static variables. If I had a centralized caching service, e.g., Redis or Memcache, I could add very similar functionality where I check the variable in the centralized cache instead.

Unit Testing

I briefly mentioned this in the previous post as well, by separating my functionality it makes unit testing my code much easier. I can test smaller pieces of my code, add several different test scenarios to some of the critical paths and less on the non-critical paths. For example, if I'm leveraging caching, it's very important to test this feature thoroughly and ensure my data is being refreshed appropriately as well as not being constantly queried against unnecessarily.

A few changes are required to the original code to accomplish this. An interface is required for `BlogRepo`. The constructor of `BlogRepo` requires subtle changes as well. It should be updated to accept an interface so I can fake the `DbContext` so my unit tests don't connect to a real database.

Collapse | Copy Code

```
public interface IBlogRepo : IDisposable
{
    void AddBlog(Blog blog);
    int SaveChanges();
    IOrderedQueryable<Blog> GetAllBlogsOrderedByName();
}
```

Interface for the `DbContext` and minor updates to `BloggingContext`:

Collapse | Copy Code

```
public interface IBloggingContext : IDisposable
{
    IDbSet<Blog> Blogs { get; set; }
    IDbSet<Post> Posts { get; set; }
    int SaveChanges();
}
```

```
public class BloggingContext : DbContext, IBloggingContext
{
    public IDbSet<Blog> Blogs { get; set; }
    public IDbSet<Post> Posts { get; set; }
}
```

Now the constructor in `BlogRepo` can be updated to expect an `IBloggingContext`. We'll also perform another enhancement by creating an empty constructor that will create a new `BloggingContext`:

Collapse | Copy Code

```
public class BlogRepo : IBlogRepo
{
    private readonly IBloggingContext _bloggingContext;
    public BlogRepo() : this(new BloggingContext())
    {
    }
    public BlogRepo(IBloggingContext bloggingContext)
    {
        _bloggingContext = bloggingContext;
    }
    public void Dispose()
    {
        _bloggingContext.Dispose();
    }
    …
}
```

As some of the comments alluded to from the original post, I was still creating the `DbContext` in the main *Program.cs*. By altering the `BlogRepo` and interface to implement `IDisposable`, I can either automatically create a new `BloggingContext` or pass in an already created context that will get disposed automatically at the end of execution.

Now our Main function can remove the using statement that creates a new `BloggingContext`:

Collapse | Copy Code

```
static void Main(string[] args)
{
    // Create new repo class
    BlogRepo repo = new BlogRepo();
    …
}
```

Now that I have improved the core code, I can create and add a new test project to my solution. Inside the test project, I need to create a new class called `FakeDbSet` and `FakeDbContext`. These classes will be used in my testing to mock my `BloggingContext`:

Collapse | Copy Code

```
public class FakeDbSet<T> : IDbSet<T> where T : class
{
    ObservableCollection<T> _data;
    IQueryable _query;
    public FakeDbSet()
    {
        _data = new ObservableCollection<T>();
        _query = _data.AsQueryable();
    }
    public T Find(params object[] keyValues)
    {
```

```
        throw new NotSupportedException("FakeDbSet does not support the find operation");
    }
    public T Add(T item)
    {
        _data.Add(item);
        return item;
    }
    public T Remove(T item)
    {
        _data.Remove(item);
        return item;
    }
    public T Attach(T item)
    {
        _data.Add(item);
        return item;
    }
    public T Detach(T item)
    {
        _data.Remove(item);
        returnreturn item;
    }

}
```

And FakeDbContext which implements IBloggingContext interface:

Collapse | Copy Code

```
class FakeDbContext : IBloggingContext
{
    public FakeDbContext()
    {
        Blogs = new FakeDbSet<Blog>();
        Posts = new FakeDbSet<Post>();
    }
    public IDbSet<Blog> Blogs { get; set; }
    public IDbSet<Post> Posts { get; set; }
    public int SaveChanges()
    {
        return 0;
    }
    public void Dispose()
    {
        throw new NotImplementedException();
    }
}
```

And finally a BlogRepoTest class that tests the AddBlog and GetAllBlogsOrderedByName functions:

Collapse | Copy Code

```
/// <summary>
///This is a test class for BlogRepoTest and is intended
///to contain all BlogRepoTest Unit Tests
///</summary>
[TestFixture]
public class BlogRepoTest
{
    private BlogRepo _target;
    private FakeDbContext _context;
    #region Setup / Teardown
    [SetUp]
    public void Setup()
```

```
    {
        _context = new FakeDbContext();
        _target = new BlogRepo(_context);
    }
    #endregion Setup / Teardown
    /// <summary>
    ///A test for AddBlog
    ///</summary>
    [Test]
    public void AddBlogTest()
    {
        Blog expected = new Blog {Name = "Hello"};
        _target.AddBlog(expected);
        Blog actual = _context.Blogs.First();
        Assert.AreEqual(expected, actual);
    }
    /// <summary>
    ///A test for GetAllBlogsOrderedByName
    ///</summary>
    [Test]
    public void GetAllBlogsOrderedByNameTest()
    {
        FakeDbSet<Blog> blogs = new FakeDbSet<Blog>();
        IOrderedQueryable<Blog> expected = blogs.OrderBy(b => b.Name);
        IOrderedQueryable<Blog> actual = _target.GetAllBlogsOrderedByName();
        Assert.AreEqual(expected, actual);
    }
}
```

Interchangeable Data Access Points

In the original post, I mentioned about swapping out ORMs, this of course is not a regular scenario but could happen one day.

The other more likely scenario is multiple or different end points for the repository layer. There might be a reason for a repository to want to connect both to a database and some static files.

By creating the repository layer, the program itself that uses the data doesn't need to care or concern itself with where the day is coming from, just that it comes back as a LINQ object for further use.

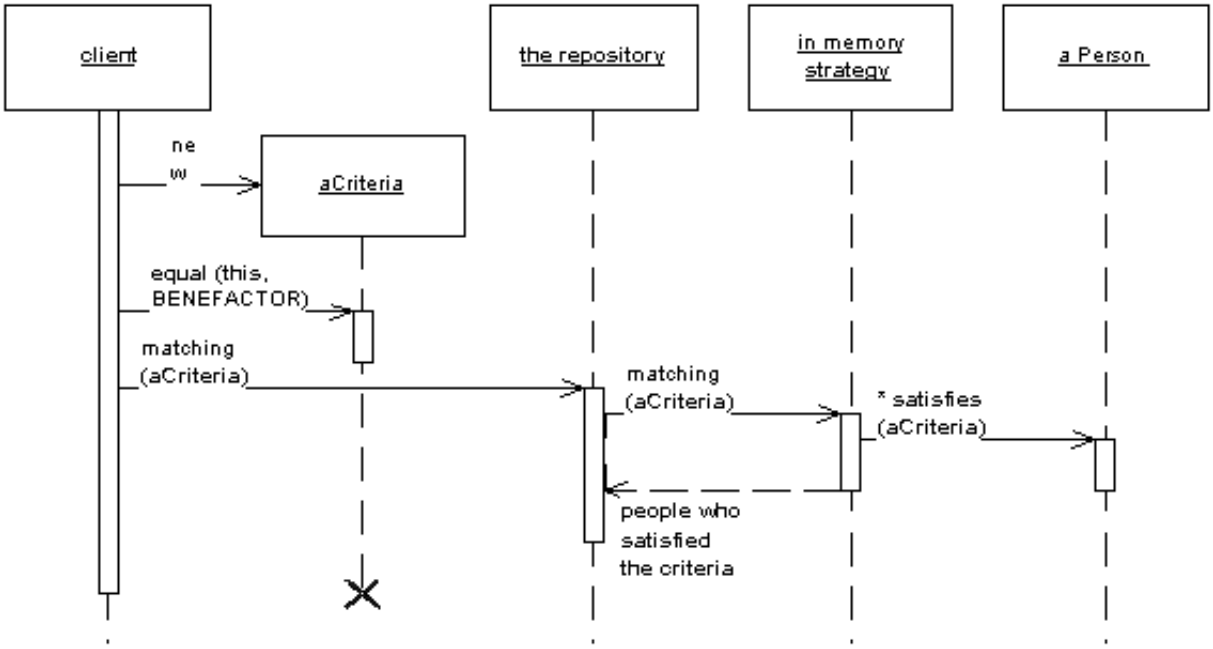
Summary

This probably might just be a tipping point for many of the great reasons to use a repository pattern. Hopefully this posts helps fill in some of the gaps from the original post on Entity Framework Beginner's Guide Done Right. Once again, the full source is available on GitHub: https://github.com/endyourif/RepoTest/.

Repository

by Edward Hieatt and Rob Mee

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.



A system with a complex domain model often benefits from a layer, such as the one provided by Data Mapper (165), that isolates domain objects from details of the database access code. In such systems it can be worthwhile to build another layer of abstraction over the mapping layer where query construction code is concentrated. This becomes more important when there are a large number of domain classes or heavy querying. In these cases particularly, adding this layer helps minimize duplicate query logic.

A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

You can also find a good write-up of this pattern in Domain Driven Design.

Repository vs DAO

By Mike on 1 November 2012

You're right, it's confusing! You have the repository which abstracts the persistence access details and you have the Data Access Object (DAO) which used to .. abstract persistence access details. So, is there a difference between those two, are they the same thing with different name?

Well, once again the devil is in the details. In many apps, especially data-centric, DAO and a repository are interchangeable as they do the same job. But the difference starts to show up when you have more complex apps with complex business behavior. While the Repository and DAO will strict abstract the data access they have different intentions in mind.

A DAO is much closer to the underlying storage , it's really data centric. That's why in many cases you'll have DAOs matching db tables or views 1 on 1. A DAO allows for a simpler way to get data from a storage, hiding the ugly queries. But the important part is that they return data as in **object state**.

A repository sits at a higher level. It deals with data too and hides queries and all that but, a repository deals with **business/domain objects**. That's the difference. A repository will use a DAO to get the data from the storage and uses that data to restore a business object. Or it will take a business object and extract the data that will be persisted. If you have an anemic domain, the repository will be just a DAO. Also when dealing with querying stuff, most of the time a specialized query repository is just a DAO sending DTOs to a higher layer.

Recently, someone told me that my repository approach is just a badly named DAO and it doesn't match Martin Fowler's definition of the Repo. Well, besides what I've said above, there is something else when dealing with the repository pattern. I tend to apply a pattern according to its spirit and intention, not as a dogma.

We have the definition : "**Mediates between the domain and data mapping layers** using a collection-like interface for accessing domain objects. A system with a complex domain model often benefits from a layer [...] that **isolates domain objects** from details of the database access code".

So the intention of the repository is to isolate the domain objects from the database access concerns. This is the important part. The fact that the repository is using a collection-like interface, doesn't mean you MUST treat it as a collection and you MUST have ONLY Add/Remove/Get/Find functionality. Really, it's not a dogma and you don't have to apply it by the letter.

Let's think a bit. The pattern is used to acheive separation of concerns and to simplify things. If you force yourself to use a collection and a criteria and some unit of work (things that made ORMs very popular) in probably 99% of cases you just complicate things. And it's quite ironic that people are using ORMs or some other Unit of Work approach to apply the repository pattern forcing their domain objects to include data access related stuff (like making properties virtual for NHibernate or needing to provide a parameterless constructor etc). What's the point in using the repository pattern if you couple your domain objects to data access details?

Back to Repository and DAO, in conclusion, they have similar intentions only that the Repository is a higher level concept dealing directly with business/domain objects, while DAO is more lower level, closer to the database/storage dealing only with data. A (micro)ORM is a DAO that is used by a Repository. For data-centric apps, a repository and DAO are interchangeable because the 'business' objects are simple data.

The Repository Pattern Example in C#

The Repository Pattern is a common construct to avoid duplication of data access logic throughout our application. This includes direct access to a database, ORM, WCF dataservices, xml files and so on. The sole purpose of the repository is to hide the nitty gritty details of accessing the data. We can easily query the repository for data objects, without having to know how to provide things like a connection string. The repository behaves like a freely available in-memory data collection to which we can add, delete and update objects.

The Repository pattern adds a separation layer between the data and domain layers of an application. It also makes the data access parts of an application better testable.

> You can download or view the solution sources on GitHub:
> LINQ to SQL version (the code from this example)
> Entity Framework code first version (added at the end of this post)

The example below show an interface of a generic repository of type T, which is a LINQ to SQL entity. It provides a basic interface with operations like Insert, Delete, GetById and GetAll. The SearchFor operation takes a lambda expression predicate to query for a specific entity.

```
using System;
using System.Linq;
using System.Linq.Expressions;

namespace Remondo.Database.Repositories
{
    public interface IRepository<T>
    {
        void Insert(T entity);
        void Delete(T entity);
        IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate);
        IQueryable<T> GetAll();
        T GetById(int id);
    }
}
```

The implementation of the IRepository interface is pretty straight forward. In the constructor we retrieve the repository entity by calling the datacontext GetTable(of type T) method. The resulting Table(of type T) is the entity table we work with in the rest of the class methods. e.g. SearchFor() simply calls the Where operator on the table with the predicate provided.

```
using System;
using System.Data.Linq;
using System.Linq;
using System.Linq.Expressions;

namespace Remondo.Database.Repositories
{
    public class Repository<T> : IRepository<T> where T : class, IEntity
    {
        protected Table<T> DataTable;

        public Repository(DataContext dataContext)
        {
            DataTable = dataContext.GetTable<T>();
        }

        #region IRepository<T> Members

        public void Insert(T entity)
        {
            DataTable.InsertOnSubmit(entity);
        }

        public void Delete(T entity)
        {
            DataTable.DeleteOnSubmit(entity);
        }

        public IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate)
        {
            return DataTable.Where(predicate);
        }

        public IQueryable<T> GetAll()
        {
            return DataTable;
        }

        public T GetById(int id)
        {
            // Sidenote: the == operator throws NotSupported Exception!
            // 'The Mapping of Interface Member is not supported'
            // Use .Equals() instead
            return DataTable.Single(e => e.ID.Equals(id));
        }

        #endregion
    }
}
```

The generic GetById() method explicitly needs all our entities to implement the IEntity interface. This is because we need them to provide us with an Id property to make our generic search for a specific Id possible.

```
namespace Remondo.Database
{
    public interface IEntity
    {
        int ID { get; }
```

```
    }
}
```

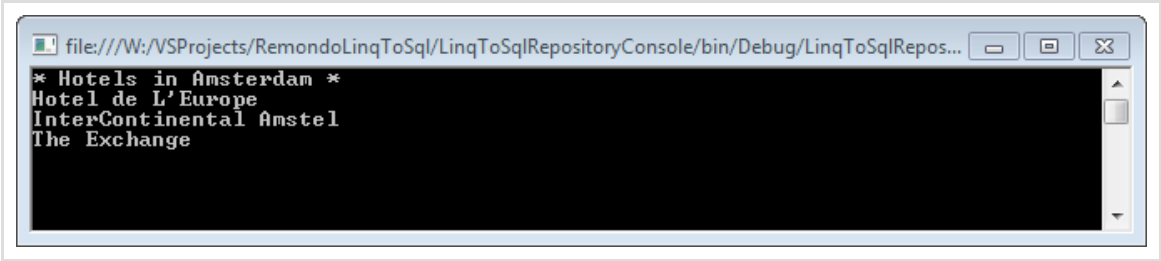
Since we already have LINQ to SQL entities with an Id property, declaring the IEntity interface is sufficient. Since these are partial classes, they will not be overridden by LINQ to SQL code generation tools.

```
namespace Remondo.Database
{
    partial class City : IEntity
    {
    }

    partial class Hotel : IEntity
    {
    }
}
```

We are now ready to use the generic repository in an application.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Remondo.Database;
using Remondo.Database.Repositories;

namespace LinqToSqlRepositoryConsole
{
    internal class Program
    {
        private static void Main()
        {
            using (var dataContext = new HotelsDataContext())
            {
                var hotelRepository = new Repository<Hotel>(dataContext);
                var cityRepository = new Repository<City>(dataContext);

                City city = cityRepository
                    .SearchFor(c => c.Name.StartsWith("Ams"))
                    .Single();

                IEnumerable<Hotel> orderedHotels = hotelRepository
                    .GetAll()
                    .Where(c => c.City.Equals(city))
                    .OrderBy(h => h.Name);

                Console.WriteLine("* Hotels in {0} *", city.Name);

                foreach (Hotel orderedHotel in orderedHotels)
                {
                    Console.WriteLine(orderedHotel.Name);
                }

                Console.ReadKey();
            }
        }
    }
```

```
}
```



Once we get of the generic path into more entity specific operations we can create an implementation for that entity based on the generic version. In the example below we construct a HotelRepository with an entity specific GetHotelsByCity() method. You get the idea. ;-)

```
using System.Data.Linq;
using System.Linq;

namespace Remondo.Database.Repositories
{
    public class HotelRepository : Repository<Hotel>, IHotelRepository
    {
        public HotelRepository(DataContext dataContext)
            : base(dataContext)
        {
        }

        public IQueryable<Hotel> FindHotelsByCity(City city)
        {
            return DataTable.Where(h => h.City.Equals(city));
        }
    }
}
```

The Specification Pattern: A Primer

Posted March 25th 2005 by Matt Berther

The Specification pattern is a very powerful design pattern which can be used to remove a lot of cruft from a class's interface while decreasing coupling and increasing extensibility. It's primary use is to select a subset of objects based on some criteria, and to refresh the selection at various times.

For example, I've seen a lot of classes that have interfaces that look something similar to this:

```
public class User
{
    public string Company;
    public string Name;
    public string City;
}

public class UserProvider
{
    public User[] GetUserByName(string name)
    {
    }

    public User[] GetUsersByCity(string name)
    {
    }

    public User[] GetUsersByCompany(string company)
    {
    }
}
```

Using this model, you can see that every time you want to add a new condition for user retrieval, you have to add a method to the UserProvider class which obfuscates the interface.

Now, lets look at the same example using the specification pattern.

```
public class User
{
    public string Company;
    public string Name;
    public string City;
}

public class UserSpecification
{
    public virtual bool IsSatisfiedBy(User user)
    {
        return true;
    }
}

public class UserProvider
{
    public User[] GetBySpecification(UserSpecification spec)
    {
        ArrayList list = new ArrayList();

        UserCollection coll = SomeMethodToPopulateTheUserCollection();
        foreach (User user in coll)
        {
            if (spec.IsSatisfiedBy(user))
            {
                list.Add(user);
            }
        }
```

```
        return (User[])list.ToArray(typeof(User));
    }
}

class UserCompanySpecification : UserSpecification
{
    private readonly string companyName;

    public UserCompanySpecification(string companyName)
    {
        this.companyName = companyName;
    }

    public override bool IsSatisfiedBy(User user)
    {
        return user.Company.Equals(companyName);
    }
}
```

Using the specification pattern, we have removed all of the specialized methods from the UserProvider class. Also, because of the loose coupling, any time we want an additional condition for user retrieval, we need to only implement a new UserSpecification and pass this instance off to the GetBySpecification method, rather than polluting the existing interface.

This allows the calling code to determine exactly how it wants to filter any given collection, rather than the provider code assuming that it knows how the user wants it.

Of course, there is nothing preventing an API designer from putting a few commonly used specifications into the API itself.

This pattern is very powerful, but like anything can be overused. Make sure to review the consequences in the linked description of the pattern for when you should and shouldnt use this pattern.

The Specification Pattern

Hot on the heels of my devastatingly fantastic post on an implementation of the Snapshot Pattern, I give you my next *piece du resistance*. In this little post, I'd like to delve into the Specification

So what the heck is it? Matt Berther provided a pretty good introduction where he states:

> It's primary use is to select a subset of objects based on some criteria...

That pretty much sums it up. What we want to do is extract out a *specification* for a subset of objects we might be interested in. We do this by creating specification objects.

You'll see something like this in a lot of applications:

view plain copy to clipboard print ?

```
01. public List GetHighPricedSaleProducts(){
02.   List list = new ArrayList();
03.   for(Product p in Products){
04.     if(p.IsOnSale && p.Price > 100.0){
05.       list.Add(p);
06.     }
07.   }
08.   return list;
09. }
```

This is fine in small doses, but your definition of a highly priced sale product might change over time, and we want to avoid having our logic for what *IsOnSale* and what a highly priced object act sprinkled throughout our code. One way to avoid this is to extract our logic into a Specification object like so:

view plain copy to clipboard print ?

```
01. public class ProductOnSaleSpecification : Specification<Product>
02. {
03.   public override bool IsSatisfiedBy(Product product)
04.   {
05.     return product.IsOnSale;
06.   }
07. }
```

Now the first loop I wrote can be written like so:

view plain copy to clipboard print ?

```
01. public List GetHighPricedSaleProducts(){
02.   List list = new ArrayList();
03.   for(Product p in Products){
04.     if(new ProductOnSaleSpecification().isSatisfiedBy(p)
05.       && p.Price > 100.0){
06.       list.Add(p);
07.     }
08.   }
09.   return list;
10. }
```

This is a slight improvement... There's actually more code to write, but now we can separately unit test each specification we create, without worrying about the loop:

view plain copy to clipboard print ?

```
01. [Test]
02. public void TestIsSatisfiedBy_ProductOnSaleSpecification()
03. {
04.   bool isOnSale = true;
05.   Product saleProduct = new Product(isOnSale);
06.   Product notOnSaleProduct = new Product(!isOnSale);
07.   ProductOnSaleSpecification spec =
08.     new ProductOnSaleSpecification();
09.   Assert.IsTrue(spec.IsSatisfiedBy(saleProduct));
10.   Assert.IsFalse(spec.IsSatisfiedBy(notOnSaleProduct));
11. }
```

Ok, so that's interesting, but we haven't even gone halfway, here. Why don't we refine that loop I wrote to use the new Generic collections in .NET 2.0:

view plain copy to clipboard print ?

```
01. public List<Product> GetSaleProducts(){
02.   return Products.FindAll(
03.     new ProductOnSaleSpecification().IsSatisfiedBy);
04. }
```

Wow, now there's some serious savings on lines of code. "But you're missing the bit about the high priced products from the first example!?!" I hear you saying. Fear not, let's extract that into a specification like so:

view plain copy to clipboard print ?

```
01. public class ProductPriceGreaterThanSpecification : Specification<Product>
02. {
03.     private readonly double _price;
04.     public ProductPriceGreaterThanSpecification(double price)
05.     {
06.         _price = price;
07.     }
08.     public override bool IsSatisfiedBy(Product product)
09.     {
10.         return product.Price > _price;
11.     }
12. }
```

We're still left with one problem, though. How do we tell the generic list of all products that we want the products that are *both* on sale and over a certain price? Let's try extracting our functionali Specification superclass first. This is what our *ProductOnSaleSpecification* and *ProductPriceGreaterThanSpecification* will inherit from. Once that's over with, we can create a *CompositeSpecifi*

abstract, and allows us to pass in the left and right sides of a specification "equation." We can then implement yet another subclass (this time of CompositeSpecification) that we'll call *AndSpec*
it is:

view plain copy to clipboard print ?

```
01. public class AndSpecification<T> : CompositeSpecification<T>
02. {
03.   public AndSpecification(Specification<T> leftSide,
04.     Specification<T> rightSide)
05.       : base(leftSide, rightSide)
06.   {}
07.   public override bool IsSatisfiedBy(T obj)
08.   {
09.     return _leftSide.IsSatisfiedBy(obj)
10.       && _rightSide.IsSatisfiedBy(obj);
11.   }
12. }
```

Now our original loop that looks for highly priced products that are on sale looks like this:

view plain copy to clipboard print ?

```
01. public List<Product> GetSaleProducts(){
02.   AndSpecification spec = new AndSpecification(
03.     new ProductOnSaleSpecification(),
04.     new ProductPriceGreaterThanSpecification());
05.   return Products.FindAll(spec.IsSatisfiedBy);
06. }
```

We're getting there, but we're *still* not done. The code we just wrote is soooo .NET 1.1. Let's get fluent with our interfaces and add some sweet sugar to our Specification base class...

view plain copy to clipboard print ?

```
01. public abstract class Specification<T>
02. {
03.   public abstract bool IsSatisfiedBy(T obj);
04.   public AndSpecification<T> And(Specification<T> specification)
05.   {
06.     return new AndSpecification<T>(this, specification);
07.   }
08.   public OrSpecification<T> Or(Specification<T> specification)
09.   {
10.     return new OrSpecification<T>(this, specification);
11.   }
12.   public NotSpecification<T> Not(Specification<T> specification)
13.   {
14.     return new NotSpecification<T>(this, specification);
15.   }
16. }
```

I've just added some convenience methods to Specification that will let us chain together any specifications we create. Therefore, our original loop ascends to a new level of sexiness...

view plain copy to clipboard print ?

```
01. public List<Product> GetSaleProducts(){
02.   return Products.FindAll(
03.     new ProductOnSaleSpecification().And(
04.       new ProductPriceGreaterThanSpecification(100)).IsSatisfiedBy);
05. }
```

Friday, July 22, 2011

Specifications Pattern with LINQ

Recently I was reading two books Domain Driven Design and Refactoring to Patterns. Both these books have references to and also examples related to a pattern called Specification Pattern. This pattern also finds its way in the Patterns of Enterprise Application and Architecture by Martin Fowler. Here is my attempt to adapt this pattern using the new features available in DotNet like LINQ, Lambda Expression, Generics etc.

Code without Specification Pattern in place

To keep things simple, I'll try to reuse the example in the book **Refactoring to Patterns**. This example talks about a product finder functionality using a **ProductRepository**. There are a set of products and we need to apply filter based on different criteria. It can be a single criteria like the color, price or size of the product. The product can also be searched using composite like color and price, size and price or also products which do not meet certain criteria like color. Here is a product repository having various methods to filter the products.

```
public class ProductRepositoryWithoutSpecification
    {
        private IList<Product> _products = new List<Product>();

        public void Add(Product product)
        {
            _products.Add(product);
        }

        public IList<Product> FindProductsByColor(ProductColor color)
        {
            return _products.Where(product => product.Color == color).ToList();
        }

        public IList<Product> FindProductsByPrice(double price)
        {
            return _products.Where(product => product.Price == price).ToList();
        }

        public IList<Product> FindProductsBelowPrice(double price)
        {
            return _products.Where(product => product.Price < price).ToList();
        }

        public IList<Product> FindProductsAbovePrice(double price)
        {
            return _products.Where(product => product.Price > price).ToList();
        }

        public IList<Product> FindProductsByColorAndBelowPrice(ProductColor color, double price)
        {
            return _products.Where(product => product.Color == color && product.Price < price).ToList();
        }

        public IList<Product> FindProductsByColorAndSize(ProductColor color, ProductSize size)
        {
            return _products.Where(product => product.Color == color && product.Size == size).ToList();
        }

        public IList<Product> FindProductsByColorOrSize(ProductColor color, ProductSize size)
        {
            return _products.Where(product => product.Color == color || product.Size == size).ToList();
        }

        public IList<Product> FindProductsBySizeNotEqualTo(ProductSize size)
        {
            return _products.Where(product => product.Size != size).ToList();
        }
    }
```

As can be seen from the above code, there are 8 different versions of Find methods. This is only a subset and you can extend this to different permutations and combinations of various attributes of the product which include Color, Size, Price. The domain model contains the Product class which has very minimul properties. But you can imagine a real life product which can have numerous attributes like IsInStock, IsOnPromotion etc. etc. The attributes I have defined are sufficient for time being to demonstrate the refactoring towards the Specification Pattern. Before we start with the refactoring,

I want to build a suite of test cases to verify that the filtering works as expected before and after the code has been refactored. Here is the list of test cases I have built for the above 8 filters.

```
private Product _fireTruck;

        private Product _barbieClassic;

        private Product _frisbee;

        private Product _baseball;

        private Product _toyConvertible;

        private ProductRepositoryWithoutSpecification _productRepositoryWithoutSpecification;

        [TestInitialize]
        public void Setup()
        {
            _fireTruck = new Product
            {
                Id = "f1234",
                Description = "Fire Truck",
                Color = ProductColor.RED,
                Price = 8.95,
                Size = ProductSize.MEDIUM
            };

            _barbieClassic = new Product
            {
                Id = "b7654",
                Description = "Barbie Classic",
                Color = ProductColor.YELLOW,
                Price = 15.95,
                Size = ProductSize.SMALL
            };

            _frisbee = new Product
            {
                Id = "f4321",
                Description = "Frisbee",
                Color = ProductColor.GREEN,
                Price = 9.99,
                Size = ProductSize.LARGE
            };

            _baseball = new Product
            {
                Id = "b2343",
                Description = "Baseball",
                Color = ProductColor.WHITE,
                Price = 8.95,
                Size = ProductSize.NOT_APPLICABLE
            };

            _toyConvertible = new Product
            {
                Id = "p1112",
                Description = "Toy Porsche Convertible",
                Color = ProductColor.RED,
                Price = 230,
                Size = ProductSize.NOT_APPLICABLE
            };

            _productRepositoryWithoutSpecification = new ProductRepositoryWithoutSpecification();
            _productRepositoryWithoutSpecification.Add(_fireTruck);
            _productRepositoryWithoutSpecification.Add(_barbieClassic);
            _productRepositoryWithoutSpecification.Add(_frisbee);
            _productRepositoryWithoutSpecification.Add(_baseball);
            _productRepositoryWithoutSpecification.Add(_toyConvertible);
        }

        [TestMethod]
```

```
        public void ProductRepositoryConstructorTest()
        {
            Assert.IsNotNull(_productRepositoryWithoutSpecification);
        }
```

I have added 5 different products to the repository. Each product has a different combination of attributes to ensure that we cover different scenarios covered by the filters. Each of the repository method is tested using the following tests

```
[TestMethod]
        public void FindProductsByColorTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColor(ProductColor.RED);

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsByPriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByPrice(9.99);

            Assert.AreEqual(1, filteredProducts.Count);
            Assert.AreEqual("Frisbee", filteredProducts.First().Description);
            Assert.AreEqual(9.99, filteredProducts.First().Price);
        }

        [TestMethod]
        public void FindProductsBelowPriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsBelowPrice(9);

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Baseball", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsAbovePriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsAbovePrice(9);

            Assert.AreEqual(3, filteredProducts.Count);
            Assert.AreEqual("Barbie Classic", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsByColorAndBelowPriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColorAndBelowPrice(ProductColor.GREEN, 10);

            Assert.AreEqual(1, filteredProducts.Count);
            Assert.AreEqual("Frisbee", filteredProducts.First().Description);
        }

        [TestMethod]
        public void FindProductsByColorAndSizeTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColorAndSize(ProductColor.GREEN, ProductSize.SMALL);

            Assert.AreEqual(0, filteredProducts.Count);
        }

        [TestMethod]
        public void FindProductsByColorOrSizeTest()
```

```
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColorOrSize(ProductColor.GREEN, ProductSize.SMALL);

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Barbie Classic", filteredProducts.First().Description);
            Assert.AreEqual("Frisbee", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsBySizeNotEqualToTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsBySizeNotEqualTo(ProductSize.SMALL);

            Assert.AreEqual(4, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }
```

In fact the repository is built using TDD. With all the tests passing we now have the working code. If you look carefully at the code, it is concise with precisely one line in each of the method. Each method filters the products from the collection based on single or multiple attributes. Right now we don't have filters which have more than two criteria. In real life applications you'll come across situation quite often where there are multiple filters. As the number of filters start growing the code starts to smell. Imagine a situation where the products are to be filtered based on following criteria

- Color = Green
- Price > 10
- Size = Small or Large

With every attribute added to product, the filters can grow exponentially. Very soon you might find it very difficult to maintain such code. This is where the Specifications Pattern comes to the rescue.

Specifications Pattern

A very simple definition of Specification Pattern is that it filters a subset of objects based on some criteria. The criteria is nothing but the Specification. The Specification follows the **Single Responsibility Principle**. Each specification in general filters the objects based on only one condition. Lets take an example. We'll take the first test and convert that into a specification. So our first test was to filter the products based on the Color. We can build a **ColorSpecification** which takes color to be filtered as a constructor argument.

```
[TestMethod]
        public void FindProductsByColorTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithSpecification.FindProducts(new ColorSpecification(ProductColor.RED));

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }
```

We are using the FindProducts method which returns the filtered products based on the specification. Lets look at the ColorSpecification

```
public class ColorSpecification : Specification
    {
        private readonly ProductColor _productColor;

        public ColorSpecification(ProductColor productColor)
        {
            _productColor = productColor;
        }

        public override bool IsSatisfiedBy(Product product)
        {
            return product.Color.Equals(_productColor);
        }
    }
```

The ColorSpecification has a private variable for storing the color which is initialized through the constructor. There is only one method IsSatisfiedBy which returns boolean value based on whether the product color matches with the private variables value. The base Specification is an abstract class which has only one abstract method IsSatisfiedBy. We can run the test and verify that we get the same result as before.

Similar to the ColorSPecification are other specifications which depend on a single attribute like PriceSpecification, SizeSpecification, AbovePriceSpecification and BelowPriceSpecification. So we refactor the tests related to these attributes to use these specifications as shown below

```
        [TestMethod]
        public void FindProductsByPriceTest()
```

```
    {
        IList<Product> filteredProducts =
            _productRepositoryWithSpecification.FindProducts(new PriceSpecification(9.99));

        Assert.AreEqual(1, filteredProducts.Count);
        Assert.AreEqual("Frisbee", filteredProducts.First().Description);
        Assert.AreEqual(9.99, filteredProducts.First().Price);
    }

    [TestMethod]
    public void FindProductsBelowPriceTest()
    {
        IList<Product> filteredProducts =
            _productRepositoryWithSpecification.FindProducts(new BelowPriceSpecification(9));

        Assert.AreEqual(2, filteredProducts.Count);
        Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
        Assert.AreEqual("Baseball", filteredProducts.Last().Description);
    }

    [TestMethod]
    public void FindProductsAbovePriceTest()
    {
        IList<Product> filteredProducts =
                    _productRepositoryWithSpecification.FindProducts(new AbovePriceSpecification(9));

        Assert.AreEqual(3, filteredProducts.Count);
        Assert.AreEqual("Barbie Classic", filteredProducts.First().Description);
        Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
    }
```

Composite Specifications

So far so good. Lets now look at the next test which uses composite criteria. The test tries to filter the products by Color as well as below certain price. We need a combination of **ColorSpecification** and **BelowPriceSpecification**. We could apply the ColorSpecification first and then filter the records by applying the **BelowPriceSpecification**. But that doesn't look very good.

Instead we could use the Composite design pattern to compose a specification containing multiple Specifications. Composite pattern suggests that both the composite and the single specification should have the same interface. So in our composite specification we need to have the same IsSatisfiedBy method. Lets create a **AndSpecification** which can be used to filter products using any two Specification.

```
public class AndSpecification : Specification
{
    private readonly Specification _leftSpecification;

    private readonly Specification _rightSpecification;

    public AndSpecification(Specification leftSpecification, Specification rightSpecification)
    {
        _leftSpecification = leftSpecification;
        _rightSpecification = rightSpecification;
    }

    public override bool IsSatisfiedBy(Product product)
    {
        return _leftSpecification.IsSatisfiedBy(product) && _rightSpecification.IsSatisfiedBy(product);
    }
}
```

The constructor of AndSpecification takes two Specification instances and gives the result of the logical And operation between the two of them. The **OrSpecification** is exactly similar and returns the logical Or result of the two Specifications.

The interesting case is of the **NotSpecification** which is used for negating the result of a Specification. Instead of two Specifications we need only one Specification to negate its result. So the NotSpecification looks like

```
public class NotSpecification : Specification
{
    private readonly Specification _specification;

    public NotSpecification(Specification specification)
    {
        _specification = specification;
    }
```

```
        public override bool IsSatisfiedBy(Product product)
        {
            return !_specification.IsSatisfiedBy(product);
        }
    }
```

In fact the NotSpecification is not really a composite specification as it depends on a single specification. Because it involves a logical operation it is treated bit differently. The final result of building all these single and composite specifications is that the repository method becomes very simple. We no longer need different methods for performing filtering, one method does it for all cases.

```
    public class ProductRepositoryWithSpecification
    {
        private IList<Product> _products = new List<Product>();

        public void Add(Product product)
        {
            _products.Add(product);
        }

        public IList<Product> FindProducts(Specification specification)
        {
            return _products.Where(specification.IsSatisfiedBy).ToList();
        }
    }
```

The 8 methods earlier are no longer required. We have a single FindProducts which works with Specification instance.

Conclusion

As we saw in this blog, complex filtering can be simplified using Specifications Pattern. It can be used for a single value filter as well as composite filtering. We can add new attributes to the mode (Product in this case) and without modifying the FindProducts method of the repository, we can build additional filters using new Specifications. This keeps the design simple and follows the Open Closed Principle where in the repository is closed for modification but open for extension.

As always the complete working solution is available for download.

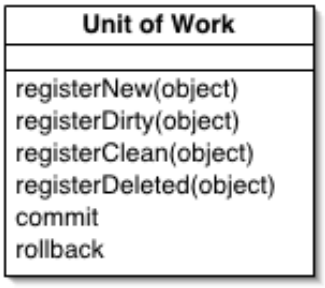
Until next time Happy Programming

Posted by nilesh at 10:42 PM +2 Recommend this on Google

Labels: AndSpecification, NotSpecification, OrSpecification, Specification, Specifications Pattern

Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.



When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.

You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests. The situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.

A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

Unit of Work Design Pattern

By **Shivprasad koirala**, 6 May 2013

What is the use of Unit of Work design pattern?

Unit of Work design pattern does two important things: first it maintains in-memory updates and second it sends these in-memory updates as one transaction to the database.

So to achieve the above goals it goes through two steps:

- It maintains lists of business objects in-memory which have been changed (inserted, updated, or deleted) during a transaction.
- Once the transaction is completed, all these updates are sent as one **big unit of work** to be persisted physically in a database in one **go**.

What is "Work" and "Unit" in a software application?

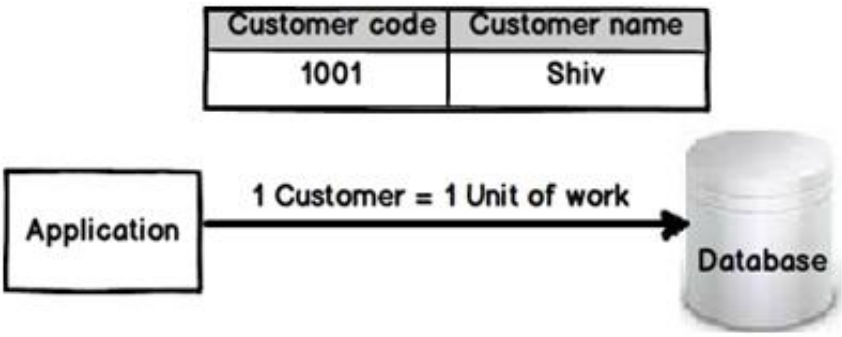
A simple definition of Work means ***performing some task***. From a software application perspective **Work** is nothing but inserting, updating, and deleting data. For instance let's say you have an application which maintains customer data into a database.

So when you add, update, or delete a customer record on the database it's one unit. In simple words the equation is.

Collapse | Copy Code

```
1 customer CRUD = 1 unit of work
```

Where CRUD stands for create, read, update, and delete operation on a single customer record.



Logical transaction! = Physical CRUD

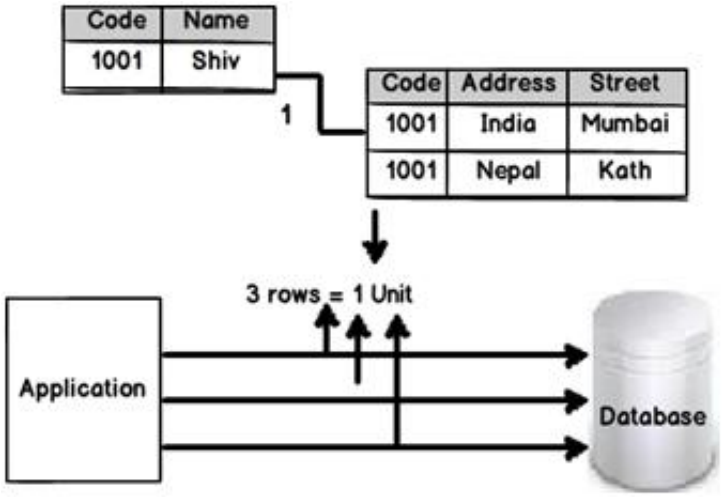
The equation which we discussed in the previous section changes a lot when it comes to real world

scenarios. Now consider the below scenario where every customer can have multiple addresses. Then many rows will become 1 unit of work.

For example you can see in the below figure customer "Shiv" has two addresses, so for the below scenario the equation is:

Collapse | Copy Code

```
3 Customer CRUD = 1 Logical unit of work
```



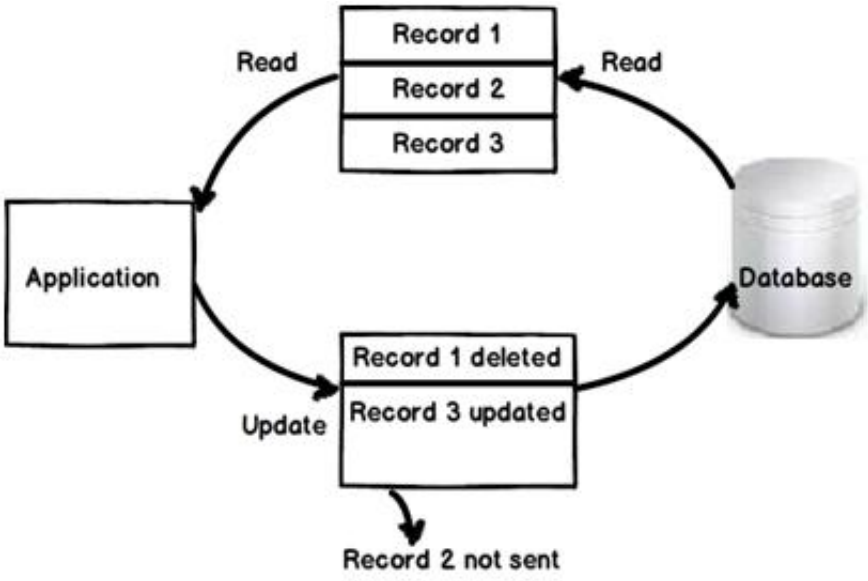
So in simple words, transactions for all of the three records should succeed or all of them should fail. It should be **ATOMIC**. In other words it's very much possible that many CRUD operations will be equal to 1 unit of work.

So this can be achieved by using simple transactions?

Many developers can conclude that the above requirement can be met by initiating all the CRUD operations in one transaction. Definitely under the cover it uses database transactions (i.e., `TransactionScope` object). But unit of work is much more than simple database transactions, it sends only changes and not all rows to the database.

Let me explain to you the same in more detail.

Let's say your application retrieves three records from the database. But it modifies only two records as shown in the below image. So only modified records are sent to the database and not all records. This optimizes the physical database trips and thus increases performance.



In simple words the final equation of unit of work is:

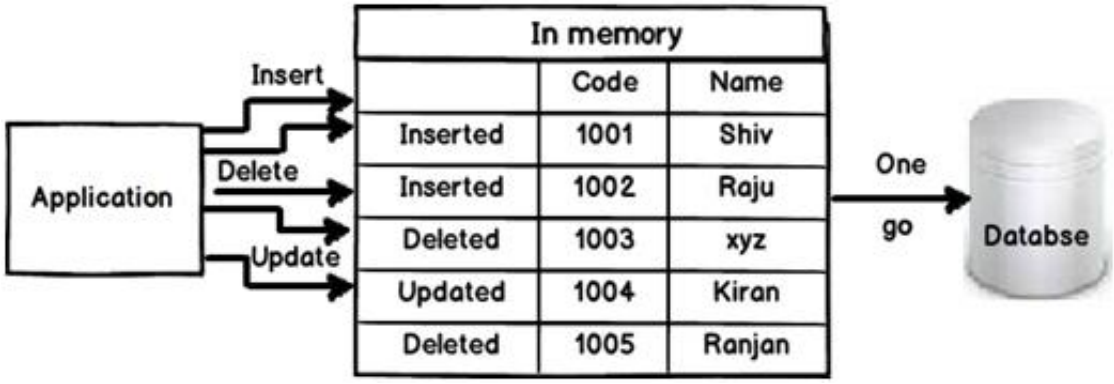
Collapse | Copy Code

```
1 Unit of work = Modified records in a transaction
```

So how can we achieve this?

To achieve the same the first thing we need is an in-memory collection. In this collection, we will add all the business objects. Now as the transaction happens in the application they will be tracked in this in-memory collection.

Once the application has completed everything it will send these changed business objects to the database in "one transaction". In other words either all of them will commit or all of them will fail.



Entity Framework Unit of Work and repositories

By Geert 27. February 2013 20:11

Catel is an application development platform for C# that was primarily focused on xaml languages (WPF, Silverlight, Windows Phone). The core however is usable by server implementations as well and more and more extensions are being developed. This blog post will give an introduction to the Unit of Work and the repositories that are available in Catel.

Overview of Unit of Work and repositories

The Repository and Unit of Work (UoW) pattern are very useful patterns to create an abstraction level over the DbContext that is provided by Entity Framework. A much heard excuse not to use repositories is that EF itself already works with repositories (the DbContext) and a UoW (in the SaveChanges method). Below are a few examples why it is a good thing to create repositories:

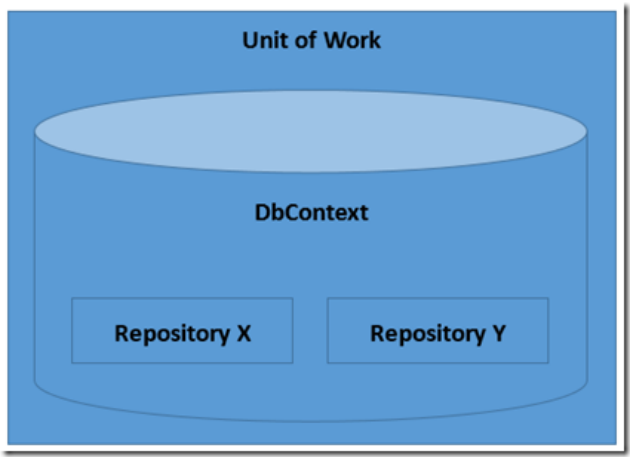
- Abstract away some of the more complex features of Entity Framework that the end-developer should not be bothered with
- Hide the actual DbContext (make it internal) to prevent misuse
- Keep security checks and saving and rollback in a single location
- Force the use of the Specification pattern on queries

A Unit of Work (UoW) is a a combination of several actions that will be grouped into a transaction. This means that either all actions inside a UoW are committed or rolled back. The advantage of using a UoW is that multiple save actions to multiple repositories can be grouped as a unit.

A repository is a class or service responsible for providing objects and allowing end-developers to query data. Instead of querying the DbContext directly, the DbContext can be abstracted away to provide default queries and force required functionality to all end-developers of the DbContext.

A Unit of Work (UoW) is a a combination of several actions that will be grouped into a transaction. This means that either all actions inside a UoW are committed or rolled back. The advantage of using a UoW is that multiple save actions to multiple repositories can be grouped as a unit.

A repository is a class or service responsible for providing objects and allowing end-developers to query data. Instead of querying the DbContext directly, the DbContext can be abstracted away to provide default queries and force required functionality to all end-developers of the DbContext.



The image above shows that the Unit of Work is the top-level component to be used. Each UoW contains its own DbContext instance. The DbContext can either be injected or will be created on the fly. Then the UoW also contains repositories which always get the DbContext injected. This way, all repositories inside a UoW share the same DbContext.

The DbContextManager

The DbContextManager class allows the sharing of DbContext (with underlying ObjectContext) classes in Entity Framework 5. The good thing about this is that the same context can be used in the same scope without having to recreate the same type of the same context over and over again.

```
using (var dbContextManager = DbContextManager<MyEntities>.GetManager())
{
    var dbContext = dbContextManager.DbContext;

    // TODO: handle logic with dbContext here
}
```

Creating a Unit of Work

A UoW can be created by simply instantiating it. The end-developer has the option to either inject the DbContext or let the DbContextManager take care of it automatically.

```
using (var uow = new UnitOfWork<MyDbContext>())
{
```

```
    // get repositories and query away
}
```

Creating a repository

A repository can be created very easily by deriving from the EntityRepositoryBase class. Below is an example of a customer repository:

```
public class CustomerRepository : EntityRepositoryBase<Customer, int>, ICustomerRepository
{
    public CustomerRepository(DbContext dbContext)
        : base(dbContext)
    {
    }
}

public interface ICustomerRepository : IEntityRepository<Customer, int>
{
}
```

Retrieving repositories from a Unit of Work

Once a UoW is created, it can be used to resolve repositories. To retrieve a repository from the UoW, the following conditions must be met:

1. The container must be registered in the *ServiceLocator* as *Transient* type. If the repository is declared as non-transient, it will be instantiated as new instance anyway.
2. The repository must have a constructor accepting a *DbContext* instance

To retrieve a new repository from the UoW, use the following code:

```
using (var uow = new UnitOfWork<MyDbContext>())
{
    var customerRepository = uow.GetRepository<ICustomerRepository>();

    // all interaction with the customer repository is applied to the unit of work
}
```

Saving a Unit of Work

It is very important to save a Unit of Work. Once the Unit of Work gets out of scope (outside the using), all changes will be discarded if not explicitly saved.

```
using (var uow = new UnitOfWork<MyDbContext>())
{
    var customerRepository = uow.GetRepository<ICustomerRepository>();

    // all interaction with the customer repository is applied to the unit of work

    uow.SaveChanges();
}
```