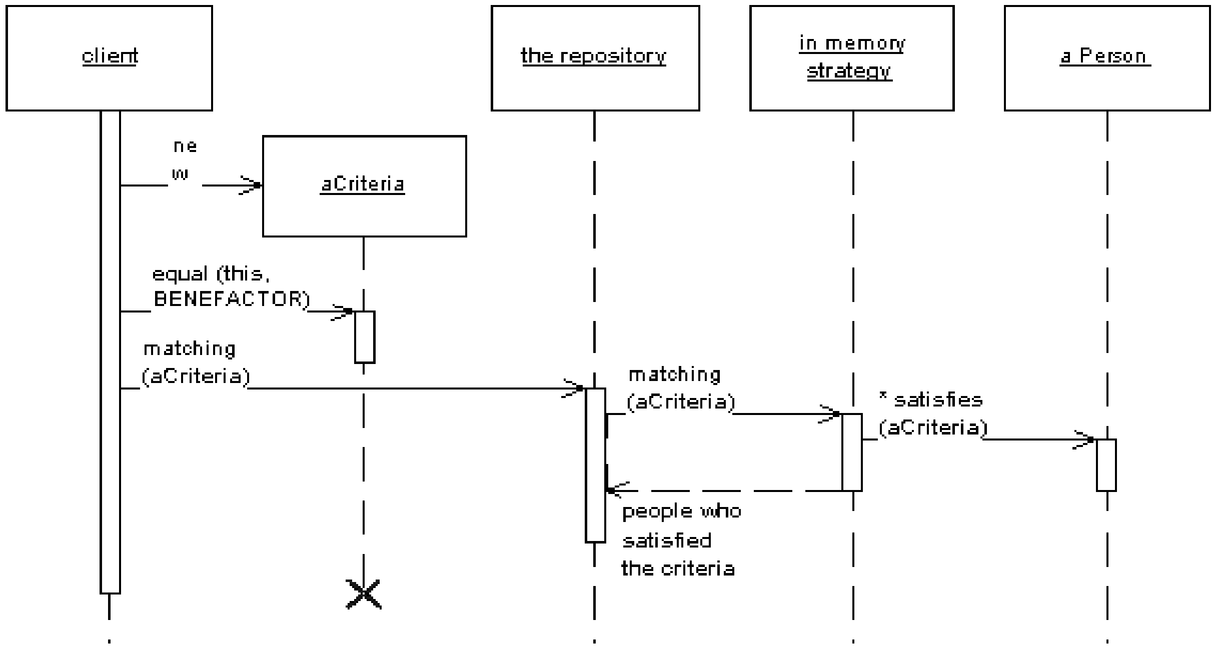
Repository

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.

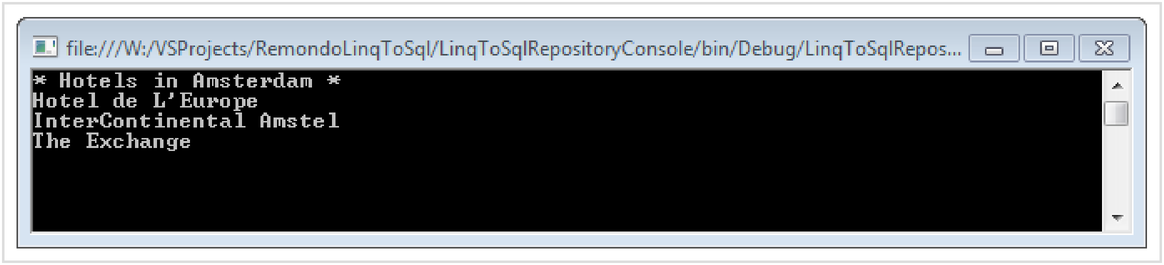
```
namespace Remondo.Database
{
    public interface IEntity
    {
        int ID { get; }
    }
}
```

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();
            }
        }
    }
}
```


file:///W:/VSProjects/RemondoLinqToSql/LinqToSqlRepositoryConsole/bin/Debug/LinqToSqlRepos...

```
* Hotels in Amsterdam *
Hotel de L'Europe
InterContinental Amstel
The Exchange
```

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));
        }
    }
}
```