

# Programming with Wolfgang

Let's talk programming

## Repository and Unit of Work Pattern

9. January 2018 by Wolfgang Ofner (<https://www.programmingwithwolfgang.com/author/wolfgang/>).

The Repository pattern and Unit of Work pattern are used together most of the time. Therefore I will combine them in this post and show how to implement them both.

### Definition Repository

The Repository mediates between the domain and data mapping layers, acting like an in-memory collection of domain objects. ("Patterns of Enterprise Application Architecture" (<http://amzn.to/2CGvtzn>) by Martin Fowler)

### Repository Pattern Goals

- Decouple Business code from data Access. As a result, the persistence Framework can be changed without a great effort
- Separation of Concerns
- Minimize duplicate query logic
- Testability

# Introduction

The Repository pattern is often used when an application performs data access operations. These operations can be on a database, Web Service or file storage. The repository encapsulates These operations so that it doesn't matter to the business logic where the operations are performed. For example, the business logic performs the method `GetAllCustomers()` and expects to get all available customers. The application doesn't care whether they are loaded from a database or web service.

The repository should look like an in-memory collection and should have generic methods like `Add`, `Remove` or `FindById`. With such generic methods, the repository can be easily reused in different applications.

Additionally to the generic repository, one or more specific repositories, which inherit from the generic repository, are implemented. These specialized repositories have methods which are needed by the application. For example, if the application is working with customers the `CustomerRepository` might have a method `GetCustomersWithHighestRevenue`.

With the data access set up, I need a way to keep track of the changes. Therefore I use the Unit of Work pattern.

## Definition Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes. ([Patterns of Enterprise Application Architecture](http://amzn.to/2CGvtzn) (<http://amzn.to/2CGvtzn>), by Martin Fowler)

© 2020 Wolfgang Richter

## Consequences of the Unit of Work Pattern

- Increases the level of abstraction and keep business logic free of data access code
- Increased maintainability, flexibility and testability
- More classes and interfaces but less duplicated code
- The business logic is further away from the data because the repository abstracts the infrastructure. This has the effect that it might be harder to optimize certain operations which are performed against the data source.

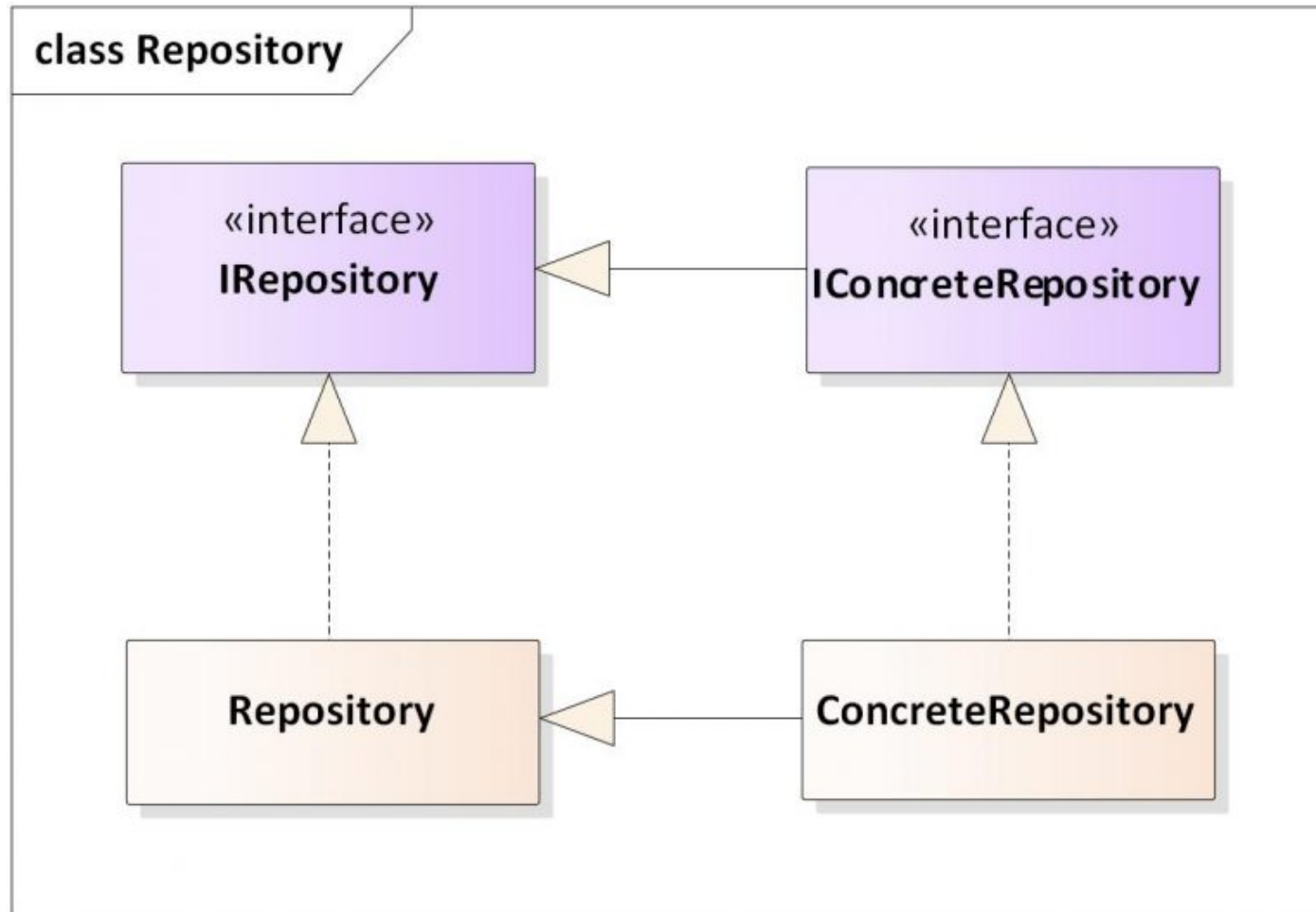
## Does Entity Framework implement the Repository and Unit of Work Pattern?

Entity Framework has a DbSet class which has Add and Remove method and therefore looks like a repository. the DbContext class has the method SaveChanges and so looks like the unit of work. Therefore I thought that it is possible to use entity framework and have all the Advantages of the Repository and Unit of Work pattern out of the box. After taking a deeper look, I realized that that's not the case.

The problem with DbSet is that its Linq statements are often big queries which are repeated all over the code. This makes to code harder to read and harder to change. Therefore it does not replace a repository.

The problem when using DbContext is that the code is tightly coupled to entity framework and therefore is really hard, if not impossible to replace it if needed.

## Repository Pattern and Unit of Work Pattern UML Diagram



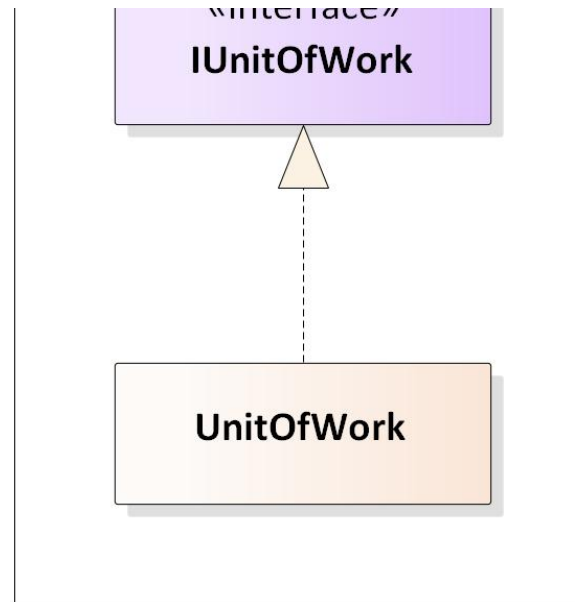
(<https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/Repository-pattern-UML-diagram.jpg>).

**Repository pattern UML diagram**

The Repository pattern consists of one IRepository which contains all generic operations like Add or Remove. It is

implemented by the Repository and by all IConcreteRepository interfaces. Every IConcreteRepository interface is implemented by one ConcreteRepository class which also derives from the Repository class. With this implementation, the ConcreteRepository has all generic methods and also the methods for the specific class. As an example: the CustomerRepository could implement a method which is called GetAllSeniorCustomers or GetBestCustomersByRevenue.





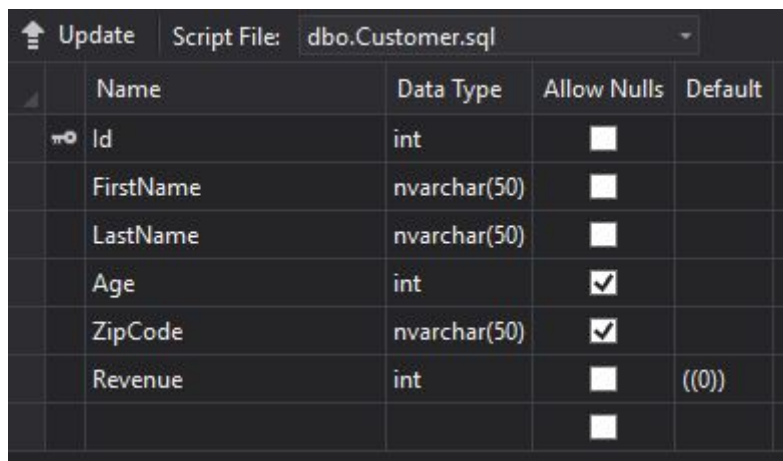
(<https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/Unit-of-Work-pattern-UML-diagram.jpg>).

The unit of work provides the ability to save the changes to the storage (whatever this storage is). The `IUnitOfWork` interface has a method for saving which is often called `Complete` and every concrete repository as property. For example, if I have the repository `ICustomerRepository` then the `IUnitOfWork` has an `ICustomerRepository` property with a getter only. Additionally, `IUnitOfWork` inherits from `IDisposable`.

The `UnitOfWork` class implements the `Complete` method, in which the data get saved to the data storage. The advantage of this implementation is that wherever you want to save something you only have to call the `Complete` method from `UnitOfWork` and don't care about where it gets saved.

# Implementation of the Repository and Unit of Work Pattern

For this example, I created a console project (RepositoryAndUnitOfWork) and a class library (RepositoryAndUnitOfWork.DataAccess). In the class library, I generate a database with a customer table.



	Name	Data Type	Allow Nulls	Default
PK	Id	int	<input type="checkbox"/>	
	FirstName	nvarchar(50)	<input type="checkbox"/>	
	LastName	nvarchar(50)	<input type="checkbox"/>	
	Age	int	<input checked="" type="checkbox"/>	
	ZipCode	nvarchar(50)	<input checked="" type="checkbox"/>	
	Revenue	int	<input type="checkbox"/>	((0))

<https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/Customer-Table.jpg>.

Next, I let Entity Framework generate the data model from the database. If you don't know how to do that, check the [documentation \(https://msdn.microsoft.com/en-us/library/jj206878\(v=vs.113\).aspx\)](https://msdn.microsoft.com/en-us/library/jj206878(v=vs.113).aspx) for a step by step walkthrough.

## Implementing Repositories

After setting up the database, it's time to implement the repository. To do that, I create a new folder, Repositories, in the class library project and add a new interface IRepository. In this Interface, I add all generic methods I want to use

later in my applications. These methods are, GetById, Add, AddRange, Remove or Find. To make the Interface usable for all classes I use the generic type parameter T, where T is a class.

```
public interface IRepository<T> where T : class
{
    1reference
    T GetById(int id);

    1reference
    IEnumerable<T> GetAll();

    1reference
    IEnumerable<T> Find(Expression<Func<T, bool>> predicate);

    1reference
    void Add(T entity);

    1reference
    void AddRange(IEnumerable<T> entities);

    1reference
    void Remove(T entity);

    1reference
    void RemoveRange(IEnumerable<T> entities);
}
```

(<https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/IRepository.jpg>).

After the generic repository, I also implement a specific repository for the customer. The ICustomerRepository inherits from IRepository and only implements one method.

```
public interface ICustomerRepository : IRepository<Customer>
```



```
{  
    1 reference  
    IEnumerable<Customer> GetBestCustomers(int amountOfCustomers);  
}
```

(<https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/ICustomerRepository.jpg>).

After implementing all interfaces it's time to implement concrete repository classes. First, I create a class Repository which inherits from IRepository. In this class, I implement all methods from the interface. Additionally to the methods, I have a constructor which takes a DbContext as Parameter. This DbContext instantiates a DbSet which will be used to get or add data.

```
public class Repository<T> : IRepository<T> where T : class  
{  
    private readonly DbSet<T> _entities;  
  
    1 reference  
    public Repository(DbContext context)  
    {  
        _entities = context.Set<T>();  
    }  
}
```

(<https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/Repository.jpg>).

The implementations of the methods are pretty straight Forward. The only interesting one might be the Find method which takes an expression as parameter. In the implementation, I use Where to find all entries which fit the Expression of the parameter.

```
1 reference
public T GetById(int id)
{
    return _entities.Find(id);
}

1 reference
public IEnumerable<T> GetAll()
{
    return _entities.ToList();
}

1 reference
public IEnumerable<T> Find(Expression<Func<T, bool>> predicate)
{
    return _entities.Where(predicate);
}
```

(<https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/Find-method.jpg>).

The final step for the Repository pattern is to implement the CustomerRepository. This class derives from Repository and ICustomerRepository and implements the method from the interface. The constructor takes a CustomerDbEntities object as Parameter which is derived from DbContext and generated by Entity Framework.

```
public class CustomerRepository : Repository<Customer>, ICustomerRepository
{
    private readonly CustomerDbEntities _customerDbEntities;

    1 reference
    public CustomerRepository(CustomerDbEntities context) : base(context)
    {
```

```
{
    _customerDbEntities = context;
}

1 reference
public IEnumerable<Customer> GetBestCustomers(int amountOfCustomers)
{
    return _customerDbEntities.Customer.OrderByDescending(x => x.Revenue).Take(amountOfCustomers).ToList();
}
```

[.https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/CustomerRepository.jpg](https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/CustomerRepository.jpg)

## Implementing Unit of Work

All repositories are created now, but I need a class which writes my data to the database, the unit of work. To implement this class, I first implement the IUnitOfWork interface in the repositories folder in the library project. This interface derives from IDisposable and has an ICustomerRepository property and the method Complete. This method is responsible for saving changes. The Name of the method could be Save, Finish or whatever you like best.

```
1 reference
public interface IUnitOfWork : IDisposable
{
    2 references
    ICustomerRepository Customers { get; }

    2 references
    int Complete();
}
```

[.https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/IUnitOfWork.jpg](https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/IUnitOfWork.jpg)

Like before, I add the concrete implementation of IUnitOfWork to the repositories folder in the console application project. The constructor takes a CustomerDbEntities object as parameter and also initializes the ICustomerRepository.

project. The constructor takes a CustomerDbEntities object as parameter and also initializes the ICustomerRepository. The Complete Method saves the context with SaveChanges and the Dispose method disposes changes.

```
public class UnitOfWork : IUnitOfWork
{
    private readonly CustomerDbEntities _context;

    1reference
    public UnitOfWork(CustomerDbEntities context)
    {
        _context = context;
        Customers = new CustomerRepository(_context);
    }

    2references
    public ICustomerRepository Customers { get; }

    2references
    public int Complete()
    {
        return _context.SaveChanges();
    }

    1reference
    public void Dispose()
    {
        _context.Dispose();
    }
}
```

[.https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/UnitOfWork.jpg](https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/UnitOfWork.jpg)

## Using the Repository and Unit of Work

The usage of the unit of work differs between a web application and a console application. In an MVC application, the unit of work gets injected into the constructor. In the console application, I have to use a using statement. I can use with

unitOfWork.Customer.Method(), for example unitOfWork.GetBestCustomers(3). To save the changes use unitOfWork.Complete().

```
using (var unitOfWork = new UnitOfWork(new CustomerDbEntities()))
{
    unitOfWork.Customers.Add(new Customer() { FirstName = "Wolfgang", LastName = "Ofner", Age = 28, ZipCode = "1234", Revenue = 9_999_999 });
    unitOfWork.Customers.Add(annoyingCustomer);
    unitOfWork.Customers.AddRange(customers);

    var foundCustomers = unitOfWork.Customers.Find(x => x.LastName == "Annoying" || x.Revenue <= 50).ToList();
    unitOfWork.Customers.Remove(foundCustomers[0]);
    foundCustomers.RemoveAt(0);
    unitOfWork.Customers.RemoveRange(foundCustomers);

    var bestCustomers = unitOfWork.Customers.GetBestCustomers(2);

    unitOfWork.Complete();
}
```

[.https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/Using-the-unit-of-work.jpg](https://www.programmingwithwolfgang.com/wp-content/uploads/2018/01/Using-the-unit-of-work.jpg)

You can find the source code on [GitHub](https://github.com/WolfgangOfner/RepositoryAndUnitOfWorkPattern) (<https://github.com/WolfgangOfner/RepositoryAndUnitOfWorkPattern>). If you want to try out the example, you have to change to connection string in the App.config to the location of the database on your computer

Note: In this example, I always talked about writing and reading data from the database. The storage location could also be a web service or file drive. If you want to try my example, you have to change the connection string for the database in the App.config file to the Location of the database on your computer.

## Conclusion

In this post, I showed how to implement the Repository and Unit of Work pattern. Implementing both patterns results in more classes but the advantages of abstraction increased testability and increased maintainability outweigh the disadvantages. I also talked about entity framework and that although it looks like an out of the box Repository and Unit of Work pattern, it comes at the cost of tight coupling to the framework and should not be used to replace the patterns.

### Related Posts:

1. **Template Method Pattern** (<https://www.programmingwithwolfgang.com/template-method-pattern/>)
2. **Visitor Pattern** (<https://www.programmingwithwolfgang.com/visitor-pattern/>)
3. **Adapter Pattern** (<https://www.programmingwithwolfgang.com/adapter-pattern/>)
4. **Facade Pattern** (<https://www.programmingwithwolfgang.com/facade-pattern/>)

## Comments

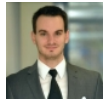


Dominic Purnell says

21. June 2018 at 9:36 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-667>).

this is really good. What is don't understand if you are creating a repository of domain objects why do we use generics and assume the underlying db entity is one and the same class as the domain object?

Reply.



Wolfgang Ofner (<https://www.programmingwithwolfgang.com>). says

25. June 2018 at 16:27 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-692>).

The generic repository contains generic operations which all entities use. The methods are generic, so every entity can use it and I don't have any problems in adding more entities later on. The repository of the domain object contains the generic methods like Save() and Delete() and also implement addition methods which only

the used entity needs.

For example the CustomerRepository implements general operations for the customer like Save(Customer) and also adds methods which are specific to the customer like GetAllCustomers().

Reply



DS says

13. March 2019 at 9:22 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-1192>).

I have a question about this part:

“The final step for the Repository pattern is to implement the CustomerRepository. This class derives from Repository and ICustomerRepository and implements the method from the interface. The constructor takes a CustomerDbEntities object as Parameter which is derived from DbContext and generated by Entity Framework.”

I don't quite understand where CustomerDbEntities comes from. Can you elaborate on how you arrived at this particular type?

Many thanks.

Reply



DS says

13. March 2019 at 9:30 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-1193>).

Never mind! On investigation of your source code, I see where this is.

Thanks.

Reply.

Pedro says

21. April 2019 at 18:28 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-1487>).

Hi, excellent post!

How do you handle the case of error with the `UnitOfWork.Complete()`? If you have to rollback the changes what will be your solution?

Bye and thanks for your time!

Reply.

Wolfgang Ofner (<https://www.programmingwithwolfgang.com>) says

24. April 2019 at 22:34 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-1495>).

Hi Pedro,

you could have a try catch in your Complete() method. If an exception occurs there, you for example log it and return 0, so the caller knows that something went wrong. Alternatively, your method which calls the Complete method could handle the exception. Though I think the first option is better.

Reply.

RepoUnit says

7. November 2019 at 20:58 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-2794>).

I really like your article. It's shorter than most other articles about this topic.

The section "Does Entity Framework implement the Repository and Unit of Work Pattern?" was really helpful for me because it confirmed my thoughts about this.

In all other articles I saw that the UnitOfWork object is a property of the Repository object. So it's exactly the other

In all other articles I saw that the UnitOfWork object is a property of the Repository object. So it's exactly the other way around than in this article. I would also let the Repository object have a property storing the UnitOfWork object since the Repository pattern intends to provide a collection-like interface to the business layer. In your solution you can only access the Repositories via the UnitOfWork which is one additional redirection. If you don't need a UnitOfWork accessing the collections should also work. What is the reason to do it like you did? In your solution you generate the DbContext object (maybe because UnitOfWork is the interface to the business layer) but the BusinessLayer should not know about implementation details (i.e. that Entity Framework is used).

Reply.

Wolfgang Ofner (<https://www.programmingwithwolfgang.com>) says  
13. November 2019 at 11:17 (<https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/#comment-2819>).

Hi RepoUnit,

The UnitOfWork object is not a property of the repository because I want to abstract the repositories behind it. This means that I can change all my repositories but don't have to make a single change in all the other layers. (if I had used dependency injection)

Yes you are right, the business layer shouldn't know anything about the UnitOfWork. I wanted to keep this demo as simple as possible. Therefore everything is in the Main method and I also don't use dependency injection.

In a real world application you would have your own business layer project and would inject the unit of work

in a real world application, you would have your own business layer project and would inject the unit of work there.

Reply

© 2020 All Rights Reserved ·