



Articles » Web Development » ASP.NET » General

Unit of Work Design Pattern

By **Shivprasad koirala**, 10 Oct 2013

★★★★★ 4.74 (27 votes)

[Download source code - Unit Work Design Pattern](#)

Table of contents

- [Are you new to Design Patterns?](#)
- [What is the use of Unit of Work design pattern?](#)
- [What is "Work" and "Unit" in a software application?](#)
- [Logical transaction! = Physical CRUD](#)
- [So this can be achieved by using simple transactions?](#)
- [So how can we achieve this?](#)
- [C# Sample code for unit of work](#)
- [Step 1: Create a generalized interface \(IEntity\) for business objects](#)
- [Step 2: Implement the "IEntity" interface](#)
- [Step 3: Create the unit of work collection](#)
- [Step 4: See it working](#)
- [Source code of Unit of work design pattern](#)

Are you new to Design Patterns?

In case you are completely new to Design Patterns you can start from the below CodeProject articles:

- [Design pattern Part 1\(factory, abstract factory, builder, prototype, shallow and deep prototype, singleton and command patterns\).](#)
- [Design pattern Part 2 \(Interpeter , Iterator, mediator, memento and observer design pattern\).](#)
- [Design pattern Part 3 \(State pattern, Strategy, Visitor, Adapter and flyweight pattern\).](#)
- [Design pattern Part 4\(Bridge Pattern, Composite Pattern, Facade Pattern, Chain Of Responsibility, Proxy Pattern and Template pattern\).](#)

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**.

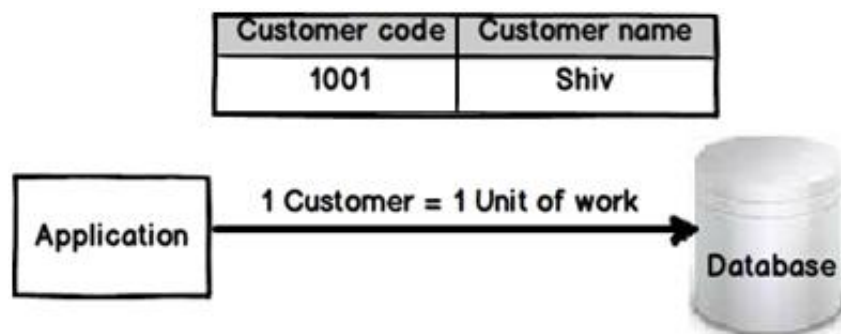
What is “Work” and “Unit” in a software application?

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.

1 customer CRUD = 1 unit of work

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

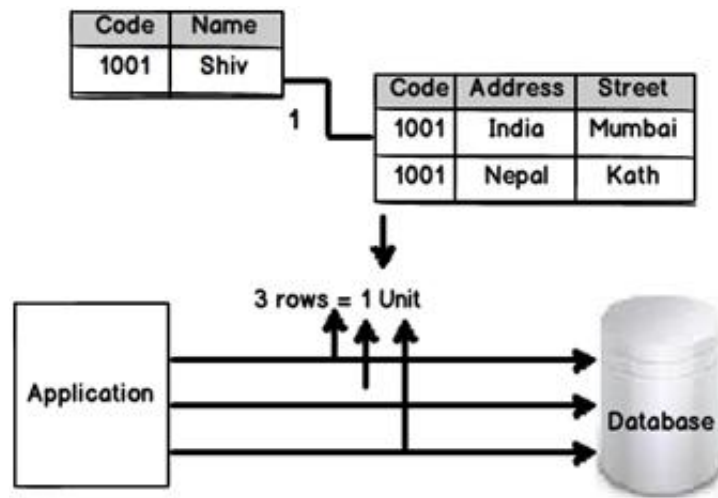


Logical transaction! = Physical CRUD

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:

3 Customer CRUD = 1 Logical unit of work



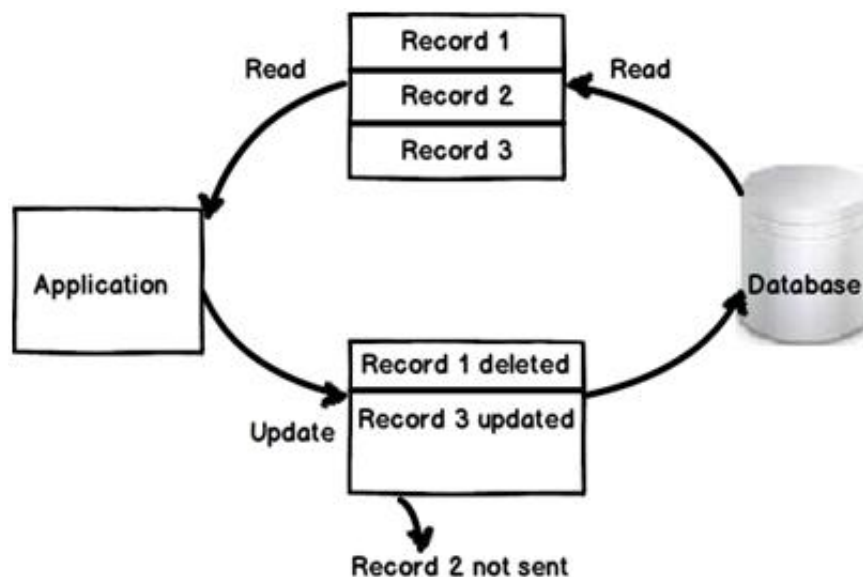
So in simple words, transactions for all of the three records should succeed or all of them should fail. It should be **ATOMIC**. In other words it's very much possible that many CRUD operations will be equal to 1 unit of work.

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.



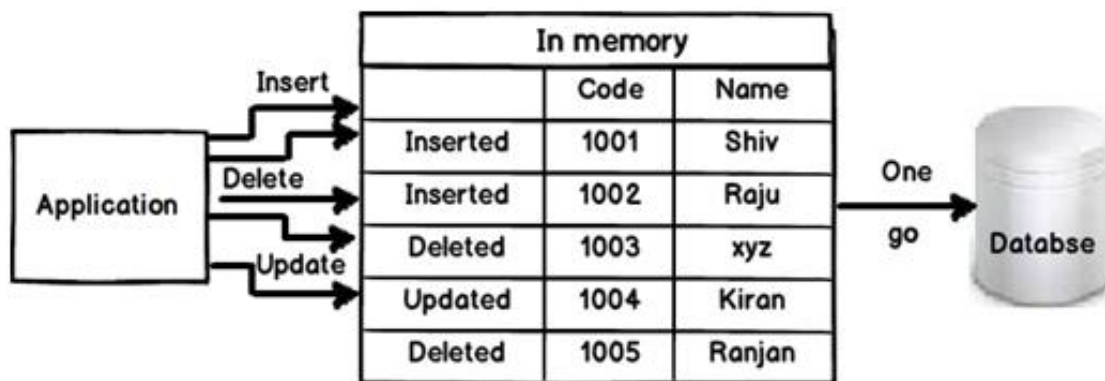
In simple words the final equation of unit of work is:

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.

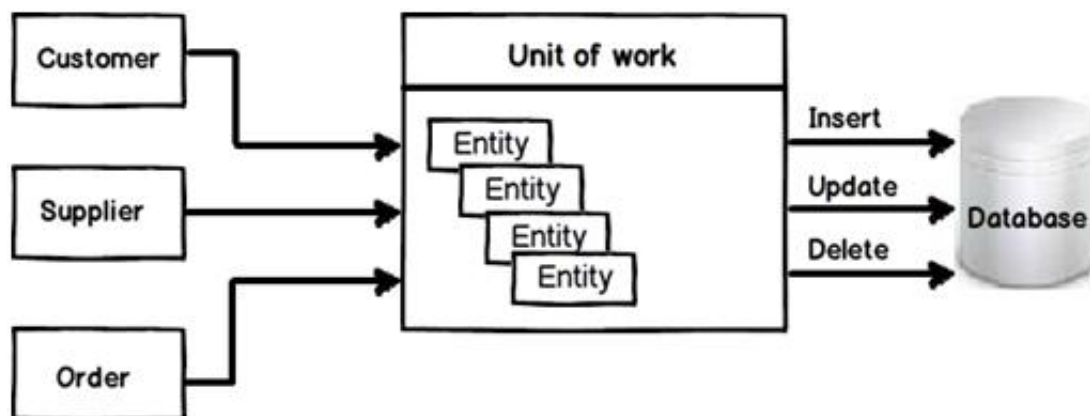
Once the application has completed everything it will send these changed business objects to the database in "one transaction". In other words either all of them will commit or all of them will fail.



C# Sample code for unit of work

Step 1: Create a generalized interface (IEntity) for business objects

At the end of the day a unit of work is nothing but a collection which maintains and track changes on the business objects. Now in an application we can have lots of business object with different types. For example you can have customer, supplier, accounts etc. In order to make the collection reusable across any business object let's generalize all business objects as just "Entities".

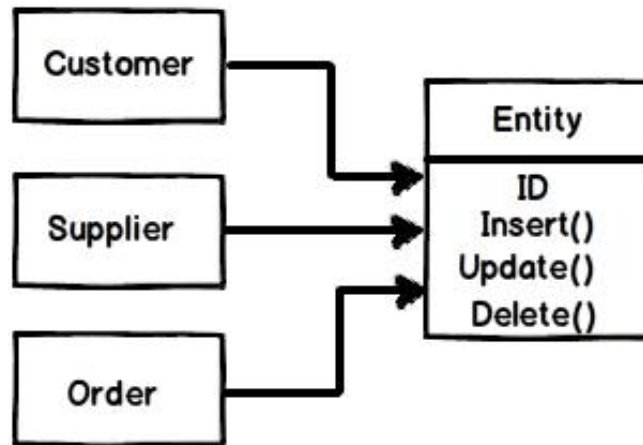


With this approach we make this collection reusable with any business object and thus avoid any kind of duplicate code.

So the first step is to create a generalized interface called **IEntity** which represents a business object in our project.

This **IEntity** interface will have an **ID** property and methods (insert, update, delete, and load) which will help us to do the CRUD operation on the business object. The **ID** property is a unique number which helps us uniquely identify the record in a database.

```
public interface IEntity
{
    int Id { set; get; }
    void Insert();
    void Update();
    List<IEntity> Load();
}
```



Step 2: Implement the IEntity interface

The next step is to implement the **IEntity** in all our business objects. For example you can see below a simple **Customer** business object which implements the **IEntity** interface. It also implements all CRUD operations. You can see that it makes a call to the data access layer to implement insert, update, and load operations.

```
public class Customer : IEntity
{
    private int _CustomerCode = 0;
    public int Id
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    private string _CustomerName = "";
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
    public void Insert()
    {
        DataAccess obj = new DataAccess();
        obj.InsertCustomer(_CustomerCode, CustomerName);
    }
    public List<IEntity> Load()
    {
        DataAccess obj = new DataAccess();
        Customer o = new Customer();
        SqlDataReader ds = obj.GetCustomer(Id);
        while (ds.Read())
        {
            o.CustomerName = ds["CustomerName"].ToString();
        }
        List<IEntity> Li = (new List<Customer>()).ToList<IEntity>();
        Li.Add((IEntity) o);
        return Li;
    }
    public void Update()
    {
        DataAccess obj = new DataAccess();
    }
```

```

        obj.UpdateCustomer(_CustomerCode, CustomerName);
    }
}

```

I am not showing the data access code in the article to avoid confusion. You can always download the full code which is attached with the article.

Step 3: Create the unit of work collection

The next step is to create the unit of work collection class. So below is a simple class we have created called **SimpleExampleUOW**.

This class has two generic collections defined as a class level variable: **Changed** and **New**. The **Changed** generic collection will store entities which are meant for updates. The **New** generic collection will have business entities which are fresh / new records.

Please note in this demo I have not implemented the delete functionality. I leave that to you as a home work 😊.

```

public class SimpleExampleUOW
{
    private List<T> Changed = new List<T>();
    private List<T> New = new List<T>();
    ....
    ....
}

```

We have also implemented the **Add** and **Load** methods which load the **New** and **Changed** collections, respectively. If the object is entered via the **Add** method it gets added to the **New** collection. If it's loaded from the database it gets loaded in the **Changed** collection.

```

public void Add(T obj)
{
    New.Add(obj);
}
public void Load(IEntity o)
{
    Changed = o.Load() as List<T>;
}

```

Now all these changes which are updated in the collection also need to be sent in **one go** to the database. So for that we have also created a **Commit** function which loops through the **New** and **Changed** collections and calls **Insert** and **Update**, respectively. We have also used the **TransactionScope** object to ensure that all these changes are committed in an atomic fashion.

Atomic fashion means either the entire changes are updated to the database or none of them are updated.

```

public void Committ()
{
    using (TransactionScope scope = new TransactionScope())
    {
        foreach (T o in Changed)
        {
            o.Update();
        }
        foreach (T o in New)
        {
            o.Insert();
        }
    }
}

```

```

        scope.Complete();
    }
}

```

Below is how the full code of **SimpleExampleUOW** looks like:

```

public class SimpleExampleUOW
{
    private List<IEntity> Changed = new List<IEntity>();
    private List<IEntity> New = new List<IEntity>();
    public void Add(IEntity obj)
    {
        New.Add(obj);
    }
    public void Committ()
    {
        using (TransactionScope scope = new TransactionScope())
        {
            foreach (IEntity o in Changed)
            {
                o.Update();
            }
            foreach (IEntity o in New)
            {
                o.Insert();
            }
            scope.Complete();
        }
    }
    public void Load(IEntity o)
    {
        Changed = o.Load() as List<IEntity>;
    }
}

```

Step 4: See it working

On the client side we can create the **Customer** object, add business objects in memory, and finally all these changes are sent in an atomic manner to the physical database by calling the **Commit** method.

```

Customer Customerobj = new Customer(); // record 1 Customer
Customerobj.Id = 1000;
Customerobj.CustomerName = "shiv";

Supplier SupplierObj = new Supplier(); // Record 2 Supplier
Supplierobj.Id = 2000;
Supplierobj.SupplierName = "xxxx";

SimpleExampleUOW UowObj = new SimpleExampleUOW();
UowObj.Add(Customerobj); // record 1 added to inmemory
UowObj.Add(Supplierobj); // record 1 added to inmemory
UowObj.Committ(); // The full inmemory collection is sent for final committ

```

Source code of Unit of Work design pattern

You can download the [source code](#) discussed in the above article.

You can also visit my site which covers more than [500 .NET and C# interview question with answers and videos](#).

Also you can see my Factory Design Pattern video here:



License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author




Shivprasad koirala

Architect <http://www.questpond.com>

India 

I am a Microsoft MVP for ASP/ASP.NET and currently a CEO of a small E-learning company in India. We are very much active in making training videos , writing books and corporate trainings. Do visit my site for [.NET, C# , design pattern , WCF , Silverlight , LINQ , ASP.NET , ADO.NET , Sharepoint , UML , SQL Server training and Interview questions and answers](#)

Comments and Discussions

 **23 messages** have been posted for this article Visit

<http://www.codeproject.com/Articles/581487/Unit-of-Work-Design-Pattern> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Mobile](#)
Web03 | 2.7.1310016.1 | Last Updated 10 Oct 2013

Article Copyright 2013 by Shivprasad **koirala**
Everything else Copyright © [CodeProject](#), 1999-2013
[Terms of Use](#)