

[articles](#) [quick answers](#) [discussions](#) [features](#) [community](#) [help](#)

Search for articles, questions, tips

[Articles](#) » [Languages](#) » [C#](#) » [General](#)

# Data Transfer Object Design Pattern in C#

**Shivprasad koirala**6 Nov 2015 [CPOL](#)Rate this:  4.80 (19 votes)

In this article we will explain about Data Transfer Object Design pattern in C#

## Table of Contents

- [Introduction and Definition](#)
- [Scenario 1 :- Improve performance](#)
- [Scenario 2:- Flatten object hierarchy](#)
- [Scenario 3:- Exclude properties](#)
- [Difference between DTO and Business objects](#)

## Introduction and Definition

DTO (Data Transfer objects) is a data container for moving data between layers. They are also termed as transfer objects. DTO is only used to pass data and does not contain any business logic. They only have simple setters and getters.



For example, below is an **Entity** class or a business class. You can see that it has business logic in the setters.

[Hide](#) [Copy Code](#)

```
class CustomerBO
{
    private string _CustomerName;
    public string CustomerName
    {
        get { return _CustomerName; }
        set
        {
            if (value.Length == 0)
            {
                throw new Exception("Customer name is required");
            }
            _CustomerName = value;
        }
    }
}
```

A data transfer object of the above **Customer** entity class would look something as shown below. It will only have setters and getters that means only data and no business logic.

[Hide](#) [Copy Code](#)

```
class CustomerDTO
{
    public string CustomerName { get; set; }
}
```

So in this article, let us try to understand in what kind of scenarios DTOs are used.

## Scenario 1 :- Improve performance

Consider the below scenario. We have a customer business class and a product business class.

[Hide](#) [Copy Code](#)

```
class CustomerBO
{
    private string _CustomerName;
    public string CustomerName
    {
        get { return _CustomerName; }
        set
        {
            if (value.Length == 0)
            {
                throw new Exception("Customer name is required");
            }
            _CustomerName = value;
        }
    }
}
```

[Hide](#) [Copy Code](#)

```
public class ProductsBO
{
    private string _ProductName;

    public string ProductName
    {
        get { return _ProductName; }
        set
        {
            if (value.Length == 0)
            {
                throw new Exception("Product Name is required");
            }
            _ProductName = value;
        }
    }

    public double ProductCost { get; set; }
}
```

Assume there is a remote client which is making calls to get **Customer** data and the **Products** they bought. Now the client has to make two calls, one to get **Customer** data and the other to get **Products** data. Now that is quite inefficient. It would be great if we can get data in one call.

[Hide](#) [Copy Code](#)

```
DataAccessLayer dal = new DataAccessLayer();  
//Call 1:- get Customer data  
CustomerBO cust = dal.getCustomer(1001);  
  
//Call 2:- get Products for the customer  
ProductsBO prod = dal.getProduct(100);
```

So to achieve the same, we can create a unified simple class which has properties from both the classes.

[Hide](#) [Copy Code](#)

```
class CustomerProductDTO  
{  
    // Customer properties  
    public string CustomerName { get; set; }  
    // Product properties  
    public string ProductName { get; set; }  
    public double ProductCost { get; set; }  
}
```

You can now get both **Customer** and **Product** data in one go.

[Hide](#) [Copy Code](#)

```
//Only one call  
CustomerProductDTO cust = dal.getCustomer(1001);
```

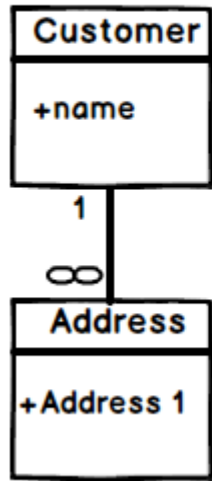
Proxy objects in WCF and Web services are good candidates for data transfer objects to improve performance.

## Scenario 2:- Flatten object hierarchy

The second scenario where I end up using DTO objects is for flattening the object hierarchy for easy data transfer.

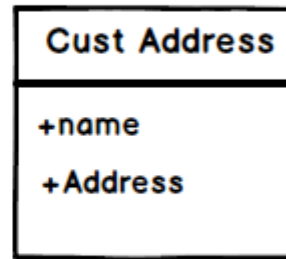
For example, you have a customer business class which has one to many relationships with an address class. But now let us say we want to write this whole data to a CSV file or transfer it across layers. In those scenarios, a flattened denormalized structure makes things easier.

### Normalized business objects



Flatten  
⇒

### Denormalized data transfer objects



So you have a **customer** class which has one to many relationship with **address** objects. It also has business rules in it.

[Hide](#) [Copy Code](#)

```

class CustomerBO
{
    // Customer properties removed for reading clarity
    public List<addressbo> Addresses { get; set; }
}
  
```

[Hide](#) [Copy Code](#)

```

class AddressBO
{
    private string _Address1;

    public string Address1
    {
        get { return _Address1; }
        set
        {
            if (value.Length == 0)
            {
                throw new Exception("Address is required");
            }
            _Address1 = value;
        }
    }
}
  
```

To create a DTO of the above **customer** and **address** class, we can just merge both the properties of the classes into one class and exclude the business logic.

[Hide](#) [Copy Code](#)

```
class CustomerDTO
{
    public string CustomerName { get; set; }
    public string ProductName { get; set; }
    public double ProductCost { get; set; }
}
```

## Scenario 3:- Exclude properties

Because DTO is completely detached from the main business object, you have the liberty to remove certain fields which you do not want to transfer to the other layers. For example, below is a simple customer business object class which has two properties called as "**IsAdmin**" and "**CustomerName**".

When you pass this business object to the UI layer, you do not wish to transfer the "**IsAdmin**" value to the UI.

[Hide](#) [Copy Code](#)

```
class CustomerBO
{
    private bool _IsAdmin;
    public string IsAdmin
    {
        get { return _IsAdmin; }
    }
    private string _CustomerName;
    public string CustomerName
    {
        get { return _CustomerName; }
        set
        {
            if (value.Length == 0)
            {
                throw new Exception("Customer name is required");
            }
            _CustomerName = value;
        }
    }
}
```

So you can create a DTO which just has the "**CustomerName**" property.

[Hide](#) [Copy Code](#)

```
class CustomerDTO
{
    public string CustomerName { get; set; }
}
```

## Difference between DTO and Business objects

	Business objects	DTO
<b>Data</b>	Yes	Yes
<b>Business logic</b>	Yes	No
<b>Structure</b>	Normalized	Normalized or Denormalized

If we can mathematically summarize:

[Hide](#) [Copy Code](#)

```
Business object = Data + Logic
DTO = Data
```

In case you want to learn design pattern, I would suggest to learn design pattern with a project. Do not learn each design pattern individually. Because when design patterns are used in project, they overlap with each other and such kind of experience can only be felt by doing an actual project and implementing design patterns on demand and naturally.

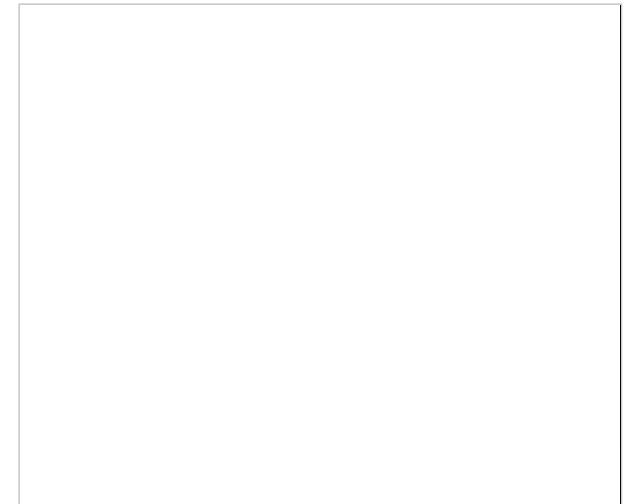
Below is a youtube video which demonstrates 15 important design pattern in a C# project:



## License

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

## Share



## About the Author

**Shivprasad koirala**





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

India

Do not forget to watch my Learn step by step video series.

[Angular Interview Questions and Answers](#)

[Learn MVC 5 step by step in 16 hours](#)

[Learn MVC Core step by step](#)

[Learn Angular tutorials step by step for beginners](#)

[Learn Azure Step by Step](#)

[Learn Data Science Step by Step](#)

[Learn Python Step by Step](#)

[Step by Step Mathematics for Data Science](#)

[Learn MSBI in 32 hours](#)

...

[show more](#)

## Comments and Discussions

You must [Sign In](#) to use this message board.



Spacing

Relaxed ▼
















Layout

Normal ▼

Per page

25 ▼

[Update](#)

 <b>Wrong in Scenario 1</b> 	 <b>Udit Gandhi</b>	<b>12-Apr-19 20:06</b>
 <b>Naming Convention</b> 	 <b>Member 14011819</b>	<b>8-Oct-18 3:20</b>
 <b>Wrong in scenario 2</b> 	 <b>Member 4549836</b>	<b>13-Jul-18 19:54</b>
 <b>flattening the object hierarchy</b> 	 <b>Mohammad Aghazadeh</b>	<b>8-Feb-18 2:43</b>
 <b>A liitle bit of deep code will be great</b> 	 <b>gemese</b>	<b>8-Apr-17 9:15</b>
 <b>Scenario 2 - last code chunk</b> 	 <b>jleol</b>	<b>29-Jul-16 0:53</b>
 <b>Token or Decorator class</b> 	 <b>Michael Breeden</b>	<b>10-Nov-15 2:22</b>
 <b>My vote of 4</b> 	 <b>dmjm-h</b>	<b>9-Nov-15 10:42</b>
 <b>Scenaio 2 example</b> 	 <b>Сергій Ярошко</b>	<b>7-Nov-15 7:33</b>
 <b>Great Article once Again from your Side</b> 	 <b>aarif moh shaikh</b>	<b>6-Nov-15 19:32</b>

Last Visit: 24-Mar-20 19:47 Last Update: 24-Mar-20 20:34

[Refresh](#)**1**

 General
  News
  Suggestion
  Question
  Bug
  Answer
  Joke
  Praise
  Rant
  Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#)[Advertise](#)[Privacy](#)[Cookies](#)[Terms of Use](#)Layout: [fixed](#) | [fluid](#)

Article Copyright 2015 by **Shivprasad koirala**  
 Everything else Copyright © [CodeProject](#), 1999-2020

Web04 2.8.200307.1