Creating the Value Objects

Now we will create the different `Command` classes, which will implement this interface and actually be responsible for executing commands. Each command might need some arguments, and might return results as well. Because we will not be using custom objects here, we need to create generic wrappers to hold the arguments and the results. We will use the `CommandArg` class to hold the arguments, and the `CommandResults` class to hold the results.

Here is the code for the `CommandArg` class:

```
[Serializable]
public class CommandArg
   {
      private NameValueCollection _paramCollection=
                                              new NameValueCollection();
      public NameValueCollection ParamCollection
      {
         get
         {
            return _paramCollection;
         }
         set
         {
            _paramCollection=value;
         }
      }
   }
```

In this `CommandArg` class, we have a `NameValueCollection (ParamCollection)`, which will wrap all of the command parameters (coming from the GUI) and their names, so that we can use them in the Service Interface layer.

Next, we will look at the `CommandResults` class, which will wrap the results after the command is executed, and then these results can be used by the UI layer.

```
[Serializable]
public class CommandResult
   {
      private object _scalarResult;
      private string _errorMessage="";

      public string ErrorMessage
      {
         get
```

```
            {
               return _errorMessage;
            }
            set
            {
               _errorMessage=value;
            }
         }
         public object ScalarResult
         {
            get
            {
               return _scalarResult;
            }
            set
            {
               _scalarResult=value;
            }
         }
      }
```

This `CommandResult` class defines two properties:

- `ErrorMessage`: to wrap any error while the command is executed by the lower layers. This error can then be displayed accordingly in the GUI.
- `ScalarResult`: this property will wrap the results of the command execution in an object and then send this object to the UI layer.

Both of these objects would be common to both the GUI and the Service Interface layers, which are used to pass data along these layers. We can also use Data Transfer Objects here.

Now, we will see what the actual `Command` class that executes a command looks like:

```
      public class GetCustomerCmd : ICommand
      {
        public CommandResult Execute(CommandArg cmdArg)
        {

           CommandResult cmdResult = CustomerServiceInterface.
                                                  GetCustomers(cmdArg);

           return cmdResult;
        }
      }
```