In the above `GetCustomerCmd` class, the `Execute` method calls the `Service interface` object (we will define this in the next section), which in turn handles the business logic calls and returns the result in the `Command Result` object. The service interface method will refer to the BL layer, and use the BL objects to return the results. Here is a sample implementation:

```
public class CustomerServiceInterface
  {
     public CommandResult GetCustomers (CommandArg cmdArg)
     {
        CommandResult cmdResult = new CommandResult();
        CustomerCollection customers = new CustomerCollection();
        cmdArg = customer.FindCustomers(cmdArg);
        return cmdResult;
     }
  }
```

The service method above uses the `CustomerCollection` class from the BL layer, and calls the `FindCustomer` method by passing in the arguments. The FindCustomer method returns a list of `Customer` records. This list is returned by wrapping it in the `CommandResult class` object.

Creating the Command Factory

How will our interface know which command to execute? For this, we take the help of the Factory design pattern, and create a `CommandFactory` class. This class takes `eventName` as an argument and, based on this string, it creates the appropriate command.

Now, we will see how the `CommandFactory` class is implemented:

```
public class CommandFactory
   {
      public static ICommand CreateCommand(string eventName)
      {
         switch(eventName)
         {
             #region Customer Related
                case "GetCustomer":
                return new GetcustomerCmd();
               //other cases
          ...
          }
        }
   }
```

Here, we are returning the actual command based on the string `EventName`. Based on this string parameter, the Factory class returns a concrete command object to the caller. To make it more flexible, instead of using strings hard-coded in the code we can put them in an XML file, and load that file to create appropriate command objects.

Tying it all up with the GUI

In the `Addcustomer.aspx` GUI page code behind the file, we can simply use the following code to create a new customer:

```
ICommand command;
        // Call Create Command on CommandFactory passing EventName
         command = CommandFactory.CreateCommand("GetCustomer");
         CommandArg carg = new CommandArg();
         CommandResults results = null;
          carg.Add("customerID", "1");
         if (command !=null)
         {
            // Call Execute Method on Command Object.
            results  = command.Execute(cmdArg);
         }
    //bind results any data control or handle it
```

So we can clearly see that the GUI doesn't know anything about the Business Layer. All it does is create an instance of the command object by passing the required event name (`GetCustomer`) and then executing that command by passing arguments. The GUI doesn't need to know who handles the command and how the internal processing takes place. It just passes the required arguments on to the `Command` class, which then executes the command by talking to the business layer interface object.

This was just one example of how we can use the Command pattern to abstract the method invocation so that the caller does not need to worry about how the command will actually be executed. The example we studied is just one of the ways in which we can implement this pattern. However, each situation will need a different implementation, based on the actual needs although the basic principle would be same.