base, which made debugging a very painstaking and time-consuming process. With improvements in the **IDE**, VS (Visual Studio) could debug such inline code script tags, and VS 2005 upwards also supported IntelliSense in ASPX pages. But mixing code and HTML was still a bad idea for the following reasons:

- **No separation of business logic, data access code, and presentation (HTML)**: It was therefore not possible to have a distributed architecture in the sense that the entire code base was monolithic in nature and could not be physically separated.
- **Code re-use**: Code cannot be re-used in other pages, whereas in code-behind files, we can call methods from a class file in many pages.
- **Source Code Control (SCC) problems**: A developer working on a file will need to block (check-out) the entire file. In the code-behind model, different developers can work on the UI and the code logic, as we have different files for each.
- **Compilation model**: Errors won't be found until the code is executed.
- **Maintenance issue**: Long-term maintenance will be an issue.

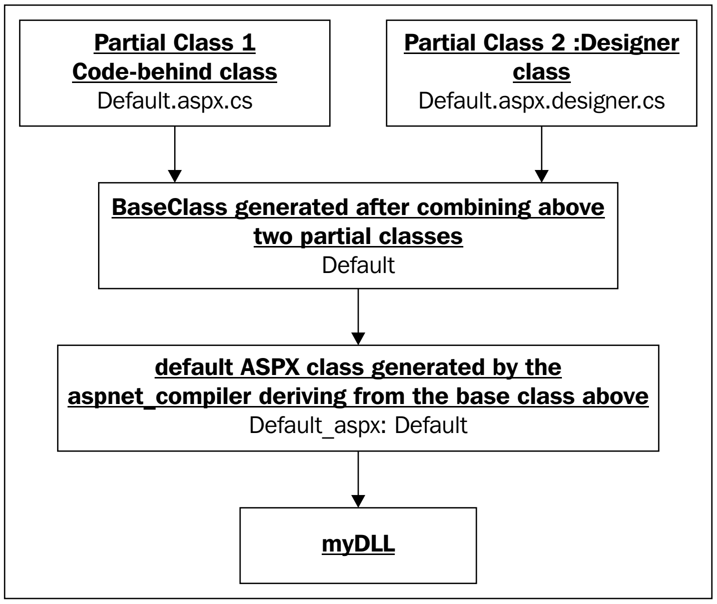
There were also some advantages in using this inline model, but the disadvantages above far outweighed any advantages:

- Because we are not using class files, simply updating a page will propagate changes to the server, without causing the users to log off, as no application restart will take place. So we can update without stopping the application, or causing an application restart.
- There can be a slight performance benefit compared to using assemblies, but this will be negligible, as modern day computing power is very fast.

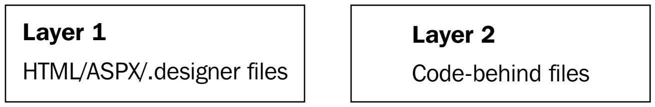
Code-Behind Model: The Second UI Layer

In the above classic ASP style example, we noticed that the code and HTML were separated but still present on the same ASPX page. ASP.NET introduced further separation using the principle of code-behind classes, by pulling all of the code out from the ASPX into a separate class and compiling it to a separate DLL. (Note that a DLL is not really required either, if the developer wishes to deploy the code-behind into the web directory. ASP.NET will compile the code "Just-In-Time" into a temporary DLL, so "pre-compiling into a DLL" is not required either.) This allowed the programmers to debug their applications more efficiently and also introduced further loose coupling in the UI layer, introducing another layer into the above 1-tier architecture.

Here is a diagrammatic representation of the above style:


Partial Class 1
Code-behind class
Default.aspx.cs
Partial Class 2 :Designer class
Default.aspx.designer.cs
BaseClass generated after combining above two partial classes
Default
default ASPX class generated by the aspnet_compiler deriving from the base class above
Default_aspx: Default
myDLL

The partial class compilation model was introduced with ASP.NET 2.0. Partial classes help us break up a main class into sibling classes, which can be merged later into one single class by the compiler. We can see that now instead of having a single ASPX file, we have three separate files for a webform—an ASPX file containing HTML UI elements, a code-behind file containing logical code, and an extra designer class file which is auto-generated by the VS and has the declaration of all of the server controls used in the ASPX form. At runtime, the code-behind class is compiled together with the `designer.cs` class (containing protected control declarations), and this merged class is used as the base class for the ASPX form class. This approach helped separate the UI code from the HTML elements, and this logical separation in terms of code-behind classes was the second layer style.


Layer 1
HTML/ASPX/.designer files
Layer 2
Code-behind files