Behavioral patterns are related to how the objects co-ordinate and communicate with each other. The Command design pattern is a famous behavioral pattern.

An important point to note is that sometimes a design pattern may use some other design patterns to solve a particular problem. So a design pattern is not atomic in nature but may comprise of multiple patterns. Also, there is no strict rule when following a design pattern, which means you don't need to copy it word-for-word, or code-by-code. A design pattern is just an approach that has worked well in similar cases for other developers. Not all situations or programming problems are the same. Hence, if we are following a particular design pattern we might need to modify it to suit our custom needs, instead of blindly following it verbatim.

There are a lot of design patterns, and discussing all of them is beyond the scope of this book. So we will focus only on a few important, basic ones that every developer must be aware of when developing web applications in ASP.NET.

Singleton Pattern

Singleton is a creational design pattern that helps us restrict and control the number of objects instantiated for a particular class throughout the application life cycle. In this section, we will understand this design pattern by examining some example code. Singleton is one of the most widely used patterns in ASP.NET.

A common programming problem is to create a single instance of an object and make sure that there are no other instances except this single instance. Only this instance should serve all incoming requests and communicate with other objects. The Singleton pattern is a design pattern that can be used to implement such scenarios.

There may be numerous programming scenarios when we may need to restrict an object to a single instance. Some of them are:

- A single instance of a mail server might be required to process all incoming mail requests.
- The `Session` object in ASP.NET is implemented using singleton pattern. That is why each user will have only one session instance accessible at any point of time.
- The `Application` object in ASP.NET is also singleton based. There is only one instance of the `Application` object for an entire application.
- We may need a single instance of a logging utility to process all logging requests in our application.

The ASP.NET framework itself implements a singleton pattern. Besides, in the `Session` object, the singleton pattern can be seen in the way a framework handles the worker process. We have one and only one instance of the work process catering to all incoming HTTP requests.

Understanding Singleton with Code Example

Let's understand the Singleton pattern with the help of an example. In our OMS application, we have many orders coming in, and we want all customers to be notified whenever they place an order. For this, we have an `EmailManager` class. This class has the responsibility to send emails to all customers who have placed orders. Now, because ASP.NET is multi-threaded (it creates a new thread for each new client request), if we start creating a new instance of the `EmailManager` (as in `EmailManager em = new EmailManager()`) to process emails, we will have a lot of such instances, for multiple requests. In simple terms, if 50 customers placed their orders through the web site, we will have 50 instances of the `EmailManager` object. So let's assume that the requirement is to handle all emails in one instance and avoid having many instances as this might have a performance impact on server memory.

Ideally, we should have a separate email application which would be handling all of these application-related emails. There can be different ways of handling emails in different applications. However, for the purpose of understanding design patterns, we will assume that we have to use Singleton to restrict the creation of `EmailManager` instances in our application.

So how can we make sure that there is one and only once instance of the `EmailManager` class throughout the life of our application? One approach is the use of **Static classes** in ASP.NET. Static classes are defined using the `static` keyword and the .NET compiler makes sure that there is no instance of a static class, and its methods can be called without creating any instance, as in:

```
public static EmailManager
{

    public static MyMethod()

   //other static methods

}
```