developerfusion.com

ASP.NET Patterns every developer should know - ASP.NET tutorial

Alex Homer

11-14 minutes

For the past year or so, I've been involved in documentation of frameworks that help developers to write better code, and to create applications that are more efficient and easier to test, debug, maintain, and extend. During that time, it has been interesting to see the continuing development of best practice techniques and tools at one of the leading software companies in our industry. Most of the work was outside my usual sphere of ASP.NET and Web development, concentrating mainly on Windows Forms applications built using .NET 2.0. This is an area where the standard design patterns that have evolved over many years are increasingly being refined and put into practice.

However, I regularly found myself wondering just how many of these patterns are equally applicable and advantageous within ASP.NET applications, where we now have the ability to write "real code" in .NET languages such as Visual Basic .NET and C# - rather than the awkward mix of script and COM components upon which classic ASP depended. Surely, out of the 250+ patterns listed on sites such as the PatternShare Community, some must be useful in ASP.NET applications. Yet a search of the Web revealed that - while there is plenty of material out there on design patterns in general, and their use in executable and Windows Forms applications - there is little that concentrates directly on the use of standard design patterns within ASP.NET.

One very useful document that is available is "Enterprise Solution Patterns Using Microsoft .NET". This discusses what design patterns are, how they are documented, and their usage in .NET Enterprise applications. It does not aim solely at ASP.NET, but has plenty of ASP.NET coverage.

About Design Patterns

Because there is so much general material available on the aims and the documentation of design patterns for executable applications, I will restrict this discussion to a few basic points before moving on to look at the patterns I find most useful in ASP.NET. The References section at the end of Part 3 of this series of articles contains links to many useful resources and Web sites related to design patterns in general.

While many people (myself included) find the term "design patterns" just a little scary - due not least to the way that they are usually described and documented - most developers use informal patterns every day when writing code. Constructs such as try...catch, using, and switch (Select Case) statements all follow standard patterns that developers have learned over time. In fact, patterns can be:

- Informal Design Patterns - such as the use of standard code constructs, best practice, well structured code, common sense, the accepted approach, and evolution over time

- Formal Design Patterns - documented with sections such as "Context", "Problem", "Solution", and a UML diagram

  Formal patterns usually have specific aims and solve specific issues, whereas informal patterns tend to provide guidance that is more general. Formal patterns usually have specific aims and solve specific issues, whereas informal patterns tend to provide guidance that is more general. Brad Appleton, author of the book "Software Configuration Management Patterns: Effective Teamwork, Practical Integration", describes design patterns, pattern languages, and the need for them in software engineering and development, like this:

  > "Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together."
  >
  > "... a body of literature to help software developers resolve recurring problems encountered throughout all of software development."
  >
  > "... a shared language for communicating insight and experience about these problems and their solutions."
  >
  > "A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns."
  >
  > (from "Patterns and Software: Essential Concepts and Terminology")

  The following sections of this article and the following two articles aim to demonstrate how you can use some of the common formal patterns (some with minor adaptations to suit ASP.NET requirements) in your Web applications to achieve the aims set out so succinctly by the Hillside Group. You can download the reasonably simple example application and open it in Visual Studio 2005, or install it to run under Internet Information Services (IIS). The download file includes a ~readme.txt file that describes the setup requirements.

Basic Design Patterns and Groups

Design patterns fall into groups, based on the type and aims of the pattern. For example, some patterns provide presentation logic for displaying specific views that make up the user interface. Others control the way that the application behaves as the user interacts with it. There are also groups of patterns that specify techniques for persisting data, define best practices for data access, and indicate optimum approaches for creating instances of objects that the application uses. The following list shows some of the most common design patterns within these groups:

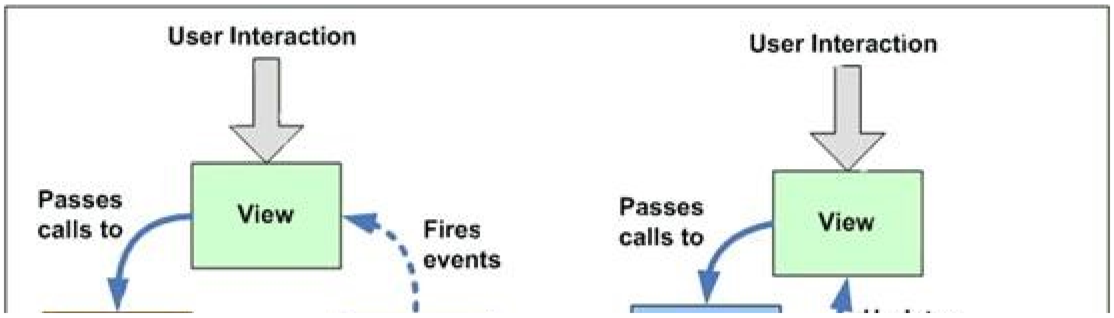
- Presentation Logic
- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Use Case Controller
- Host or Behavioral
- Command
- Publish-Subscribe / Observer
- Plug-in / Module / Intercepting Filter
- Structural
- Service Agent / Proxy / Broker

- Provider / Adapter
- Creational
- Factory / Builder / Injection
- Singleton
- Persistence
- Repository

The remaining sections of this and the two following articles discuss the patterns that are most suitable for use in ASP.NET, or which ASP.NET implements automatically and allows you to extend to adapt the behavior to suit your own requirements. See the index at the start of each article for a list of the patterns described.

The Model-View-Controller and Model-View-Presenter Patterns

The Model-View-Controller (MVC) and Model-View-Presenter (MVP) Patterns improve reusability of business logic by separating the three components required to generate and manage a specific user interface (such as a single Web page). The Model contains the data that the View (the Web page) will display and allow the user to manipulate. The Controller or Presenter links the Model and the View, and manages all interaction and processing of the data in the Model (see Figure 1).



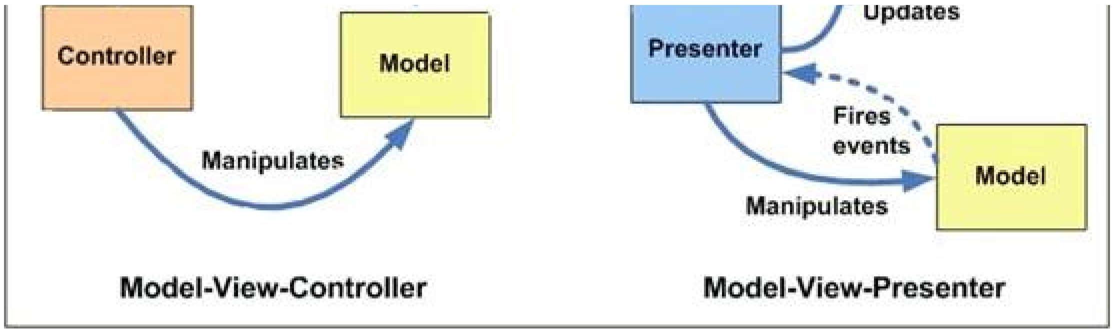


Figure 1 - The Model-View-Controller and Model-View-Presenter Patterns

In the MVC pattern, user interaction with the View raises events in the Controller, which updates the Model. The Model then raises events to update the View. However, this introduces a dependency between the Model and the View. To avoid this, the MVP pattern uses a Presenter that both updates the Model and receives update events from it, using these updates to update the View. The MVP pattern improves testability, as all the logic and processing occurs within the Presenter, but it does add some complexity to the implementation because updates must pass from the Presenter to the View.

The Provider and Adapter Patterns

The Provider and Adapter patterns allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the other. In more practical terms, these patterns provide separation between components that allows behavioral changes to occur without prior knowledge of requirements. The application and any data sources it uses, outputs it generates, or classes it must interact with, can be created independently yet still work together (see Figure 2).

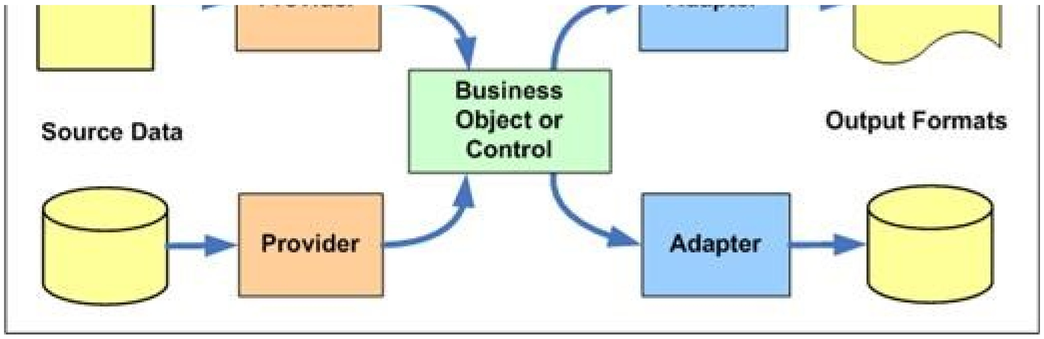


Figure 2 - The Provider and Adapter Patterns

The Provider pattern separates the data processing objects, and the application, from the source data. It allows the code in the application to be independent of the type of data source and the format of the data. A Provider component or service exposes standard methods that the application can call to read and write data. Internally, it converts these calls to the equivalents that match the data source. This means that the application can work with any source data type (such as any kind of database, XML document, disk file, or data repository) for which a suitable provider is available.

The Adapter pattern has the same advantages, and works in a similar way. Often, the target of an Adapter is some kind of output. For example, a printer driver is an example of an Adapter. ASP.NET itself, and other frameworks such as Enterprise Library, make widespread use of the Provider and Adapter patterns.

The Service Agent, Proxy, and Broker Patterns

Various patterns exist that remove dependencies between a client and a service by using intermediate brokers. There are many different implementations of the basic pattern, some of which use an extra service-agent logic component to connect the client with the local proxy or gateway interface (see Figure 3).

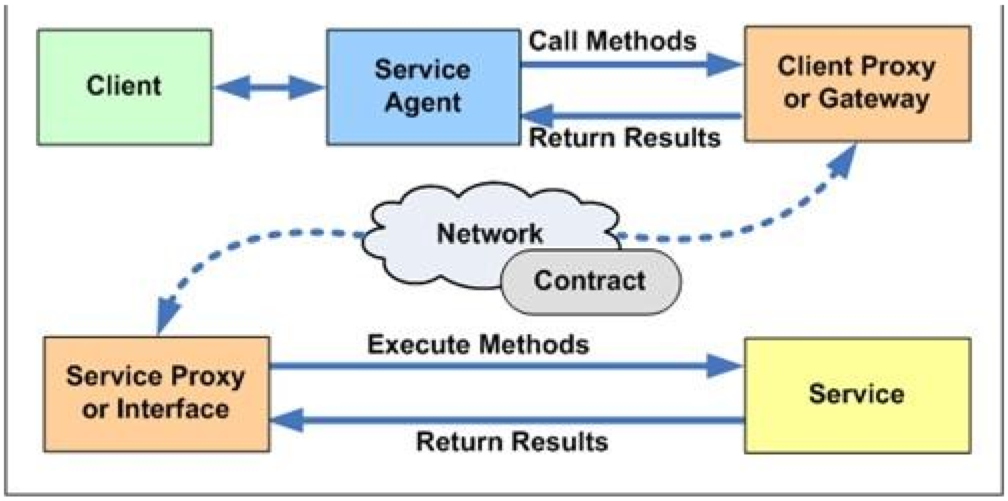


Figure 3 - The Service Agent, Proxy, and Broker Patterns

The aim of all these patterns is to allow remote connection to, and use of, a service without the client having to know how the service works. The service exposes a Contract that defines its interface, such as the Web Service Description Language (WSDL) document for a Web Service. A client-side proxy or gateway interface uses the Contract to create a suitably formatted request, and passes this to the service interface. The service sends the formatted response back through its gateway interface to the client proxy, which exposes it to the client. In effect, the client just calls the service methods on the client proxy, which returns the results just as if the service itself was a local component.

In the Service Agent pattern, an extra component on the client can perform additional processing and logic operations to further separate the client from the remote service. For example, the Service Agent may perform service address lookup, manipulate or format the client data to match the proxy requirements, or carry out any other kind of processing requirements common to different clients that use the service.

The Repository Pattern

The Repository pattern virtualizes storage of entities in a persistent medium, such as a database or as XML. For example, a repository may expose data held in the tables of a database as strongly typed Customer and Order objects rather than data sets or data rows. It effectively hides the storage implementation from the application code, and allows the use of a common set of methods in the application without requiring knowledge of the storage mechanism or format. Often, the repository uses a series of providers to connect to the source data (see Figure 4).

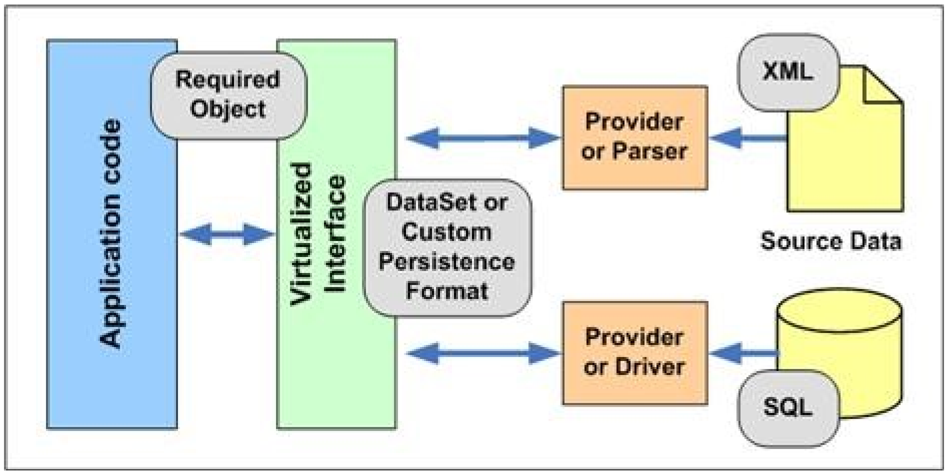


Figure 4 - The Repository Pattern

The Singleton Pattern

The Singleton pattern defines the creation of a class for which only a single instance can exist. It is useful for exposing read-only data, and static methods that do not rely on instance data. Rather than creating an instance of the class each time, the application obtains a reference to an existing instance using a static method of the class that implements the Singleton pattern.

If this is the first call to the method that returns the instance, the Singleton creates an instance, populates it with any required data, and returns that instance. Subsequent calls to the method simply return this instance. The instance lifetime is that of the application domain - in ASP.NET this is usually the lifetime of the domain Application object.