

[devx.com](https://www.devx.com)

Design Patterns for ASP.NET Developers, Part 3: Advanced Patterns – DevX

31-39 minutes



This is the last in a series of three articles that explore the built-in support for standard design patterns in ASP.NET, and ways in which you can implement common patterns in your own applications

The patterns discussed so far in the first two articles in this series ([Article 1](#), [Article 2](#)) relate to features that are either automatically implemented by ASP.NET, or are relatively easy to implement as custom features of an application. But those cover only a small proportion of the total number of design patterns, although they include those that are most useful and most commonly implemented within web applications.


The following sections discuss more advanced design patterns typically seen in Windows Forms and other desktop applications, but less often suited for implementation within web applications. However, the overall aims of these patterns?particularly the Command and Event patterns?are laudable, and it is possible to provide a useful modified implementation for ASP.NET applications. See the sidebar “Useful Resources” for a full list of

references to patterns and architectural guidance.

The Factory, Builder, and Injection Patterns

The Factory, Builder, and Injection design patterns specify various ways to separate the construction of a complex object from its representation, letting you use the same construction process to create different representations. These patterns are commonly associated with the instantiation of objects from classes, and control the way that the application receives such instances.

In the Factory pattern, subclasses of the object generator determine the object type. An example of the Factory pattern is the `DbProviderFactory` class in .NET, which generates instances of `Connection`, `Command`, and `DataAdapter` objects specific to a particular provider (such as `SqlClient` or `OleDb`) depending on the provider name you specify when you create the `DbProviderFactory` instance. Other .NET classes, such as `XmlReader` and `XmlWriter`, provide a static `Create` method that returns a pre-configured instance of the specified class.

	?
<p>Figure 1. The Command Pattern: The Command pattern separates the command invoker from the command receiver, letting developers add and alter actions and commands easily. It also enables a command to initiate multiple actions.</p>	

In the Builder and Injection patterns, the client passes instructions or hints to the object generator to specify the required object type.

In some implementations, the client can use attributes to specify the class types for injection into the current context. The object builder can set properties and execute methods on the object before returning it. It can also avoid construction overhead, and return an existing instance of an object when appropriate instead of creating a new instance. The [Microsoft patterns & practices](#) group provides a utility called ObjectBuilder that incorporates these features as part of the CAB and other software factories.

The Command Pattern

The Command pattern separates the command invoker from the command receiver, allowing developers to add commands and actions to an application without having to relate them directly. This pattern also allows a command to initiate multiple actions (see Figure 1).

The Command pattern is particularly useful in applications that load modules at runtime to extend their functionality. These modules might load in response to user interaction, or because of specific application configuration. As they load, each module can add commands to the application shell, extend the application by adding new actions, and relate the new and existing commands to new and existing actions.

The Observer and Publish-Subscribe Patterns

	?
Figure 2. The Observer and Publish-Subscribe Patterns:	

Similar to the Command pattern, but targeted toward application events, these patterns separate event creation from event consumers.	
--	--

The Observer and Publish-Subscribe patterns separate the creator of an event from the receiver(s), in much the same way as the Command pattern separates commands from actions. The main difference is that the events and actions may not relate to user commands, but to occurrences of specific events within the application (see Figure 2). For example, a data source provider may raise events when it connects to a database and executes a query. Other classes can subscribe to receive this event, allowing them to interact with the source to provide information, change the behavior, or even cancel the process.

The Observer pattern requires that the client or object wishing to receive notifications (the Observer) subscribe this interest to the object that fires the event (the Subject).

The Publish-Subscribe pattern uses an “event channel” that sits between the client or object wishing to receive notifications (the Subscriber) and the object that fires the event (the Publisher). This channel can be the .NET event system, allowing code to create application-specific events that pass a custom “arguments” instance containing values required by the subscriber. The purpose is to avoid dependencies between the Subscriber and the Publisher; therefore, unlike the standard Observer pattern, allowing any Subscriber that implements the appropriate event handler to register for and receive events generated by the Publisher.

Using Commands and Events in ASP.NET

of the classes and objects that might subscribe to these events, as described in the Publish-Subscribe pattern.

The example application demonstrates both the Publish-Subscribe pattern and a modified Observer pattern that provides a technique for executing methods (effectively, activating commands) on subscriber/observer objects that do not directly depend on the publisher/subject. Both examples implement a fictitious car production line and associated departments such as parts, sales, and transport.

In this example scenario, a client ASPX page creates an instance of the publisher (the production line) and instances of any subscribers (parts, sales, or transport departments) that should receive notification when the production line builds a new car. Each subscriber maintains a count of the number of notifications received, based on the model name of the car. The following sections of this article describe these two implementations.

Implementing the Publish-Subscribe Pattern

The example application demonstrates the Publish-Subscribe pattern by exposing a custom event from the publisher, a class named `EventProductionLine`, through an associated helper class named `BuildCarHelper`. The custom event exposes an instance of the `BuildCarEventArgs` class as its second argument (all these classes are located in the App-Code folder of the [downloadable sample code](#)). The `BuildCarEventArgs` class exposes properties that reflect the requirements of the application, and of the notification that the production line must send to subscribers. In this simple example, the two properties are the model name of the new car, and the date and time it was built:

2/24/2022, 3:57 PM

The helper class exposes a method named `Notify` that a publisher can call to notify all subscribers when it builds a new car. The code in this method first extracts any values it requires from the

EventProductionLine instance passed to it, and then creates an instance of the BuildCarEventArgs class with the appropriate values. Finally, it calls a separate method that raises the event through the .NET event system using the approach recommended by Microsoft that avoids a race condition if a subscribed class instance un-subscribes while the code is executing:

```

    public void Notify(Object publisher)    {
// raise the event to notify all subscribers
using      // the custom event arguments class
String carName = (publisher as
EventProductionLine).CurrentMotorCarName;
BuildCarEventArgs args = new BuildCarEventArgs(
carName, DateTime.Now);      // call method to
publish event using standard      // approach
defined by MS for events - see .NET docs at
// http://msdn2.microsoft.com/en-us/      //
library/w369ty8x(VS.80).aspx
OnBuildCarEvent(args);    }      // wrap event
invocations inside a protected virtual method
// to allow derived classes to override the event
invocation      // behavior      protected virtual
void OnBuildCarEvent(BuildCarEventArgs e)    {
// make a temporary copy of the event to avoid
possibility of      // a race condition if the
last subscriber un-subscribes      // immediately
after the null check and before the event      //
is raised.      EventHandler handler =
BuildCarEvent;      // event will be null if there
are no subscribers      if (handler != null)    {

```



```
// use the () operator to raise the event.
handler(this, e);    }    }
```

The following code shows the EventProductionLine class in the example application. It creates an instance of the BuildCarHelper class that it will use to raise the BuildCarEvent, and exposes it as a public property. Exposing the helper instance as a property of the EventProductionLine class means that the client application can reference the helper to subscribe to any events it exposes?in this simple example just the BuildCarEvent:

```
public class EventProductionLine    {        //
holds a reference to an instance of the helper
class that        // raises the event to notify any
subscribers        private BuildCarHelper
internalHelper = null;        // name of car
currently being built        private String carName
= String.Empty;        public
EventProductionLine()        {        // class
constructor        // get new helper class
instance        internalHelper = new
BuildCarHelper(this);        }        public
BuildCarHelper Helper        {        // property
that exposes the helper instance        get {
return internalHelper; }        }
```

The publisher will usually expose other methods that the helper class can access as it creates the custom BuildCarEventArgs instance. In this example, the only other property exposed by the EventProductionLine class returns the name of the car it is currently building.

The remaining code is the `BuildNewMotorCar` method that the client will call when it wants the production line to build a specific model of car?as specified by the parameter to this method. In the method, the code saves the name of the car in a local variable so that the `CurrentMotorCarName` property can expose it, does whatever work is required to build the car, and then calls the `Notify` method of the associated `BuildCarHelper` instance to raise the `BuildCarEvent` to all subscribed clients:

```
// any properties required to expose data
that the // subscribers may need to use.
These property values // must be wrapped
inside the custom args class that the //
Helper uses when raising the event public
String CurrentMotorCarName { get {
return carName; } } // main method for
class-this is called by // the subject to
accomplish the main tasks. There can // be
more methods, but each must call the //
Notify or other equivalent method of the helper
to // raise an event public void
BuildNewMotorCar(String motorCarName) {
// save value of car name carName =
motorCarName; // do whatever work
required here // ... // notify all
subscribers internalHelper.Notify(this);
} }
```

The subscriber implementations in the example application maintain a Hashtable in the user's session that contains a count of each new car built by the current instance of the

EventProductionLine class. In the constructor of each subscriber class, code looks for a Hashtable using the relevant session key declared in a local String constant at the start of the class file, and stores this Hashtable in a local variable. If there is no stored Hashtable, the code creates an empty one. As an example, here's the relevant code from the EventPartsDept class:

```
public class EventPartsDept {      Hashtable
partsUsed = null;      const String
SESSION_KEY_NAME = "SavedPartsCount";
public EventPartsDept() {          // class
constructor, get saved Hashtable from          //
session if available          if
(HttpContext.Current.Session[SESSION_KEY_NAME] !=
null) {          partsUsed = (Hashtable)
HttpContext.Current.Session[SESSION_KEY_NAME];
}      else {          // create new empty
Hashtable          partsUsed = new Hashtable();
}      }
```

Publish-Subscribe Implementation (continued)

Subscribers to the BuildCarEvent are not dependent on the EventProductionLine class. However, they must implement an event handler for the BuildCarEvent that the EventProductionLine class raises through its helper class. The code in that event handler can extract the values from the BuildCarEventArgs instance to get information about the event. In the following example, the code updates the Hashtable containing the count of each model built, and stores it in the session:

```
public void BuildCarEventHandler(Object
sender, BuildCarEventArgs args) {          // the
```

```

calling code links this to the BuildCarEvent so
that      // it executes automatically when a new
car is built      // code can query properties of
custom BuildCarEventArgs      // class to get
information about the event if required
String carName = args.ModelName;      // then
increment count of cars built      if
(partsUsed.ContainsKey(carName))      {
partsUsed[carName] = (Int32)partsUsed[carName] +
1;      }      else      {
partsUsed.Add(carName, 1);      }      // save
count in session
HttpContext.Current.Session[SESSION_KEY_NAME] =
partsUsed;      }

```

Of course, the subscriber classes will also expose methods and properties appropriate to their individual tasks. In the example application, they expose only the Hashtable containing the count of each car built so that the client code can display the results:

```

public Hashtable CountOfPartsUsed      {      //
expose the count of notifications as a property
get { return partsUsed; }      }

```

In the [sample application](#) the ASPX page named TransferPage4.aspx demonstrates the Publish-Subscribe implementation just described. When a user clicks the “Build Car” button, the click-handler code begins by creating an EventProductionLine class instance and instances of the three subscriber classes: EventPartsDept, EventTransportDept, and EventSalesDept (all located in the App-Code folder of the example). Here’s the code:

```

        protected void btn_Build_Click(object sender,
EventArgs e)    {           // create instance of
subject class will raise the           //
BuildCarEvent through its helper class
BuildCarHelper           EventProductionLine pLine =
new EventProductionLine();           // create
instances of required subscriber classes
EventPartsDept parts = new EventPartsDept();
EventTransportDept transport = new
EventTransportDept();           EventSalesDept sales =
new EventSalesDept();

```

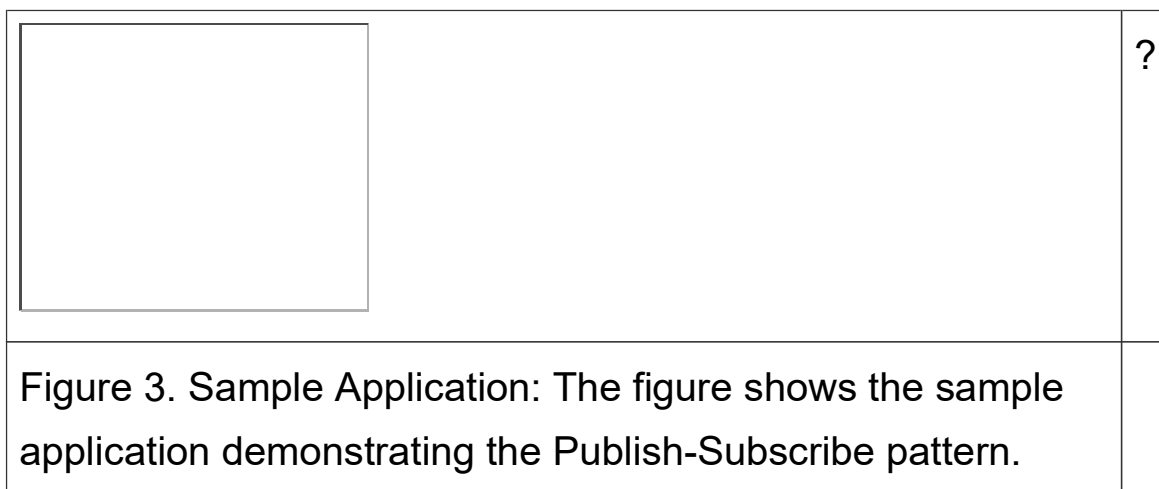
Next, the page must subscribe the three “subscriber” classes to the BuildCarEvent event. Depending on the settings of three checkboxes in the page, the code adds the BuildCarEvent handler of each subscriber class to the BuildCarEvent of the helper instance associated with the current EventProductionLine instance. Then it can call the EventProductionLine’s BuildNewMotorCar method, specifying the model name selected in the drop-down list in the ASPX page. This sends the BuildCarEvent to each subscriber. The code continues by displaying the resulting values from the Hashtable instance of each subscriber:

```

        // subscribe the appropriate instances to
the BuildCarEvent           // of the publisher (the
EventProductionLine class) according           // to
the checkbox settings in the page.           if
(chk_Parts.Checked)           {
pLine.Helper.BuildCarEvent +=
parts.BuildCarEventHandler;           }           if

```

```
(chk_Transport.Checked)      {
pLine.Helper.BuildCarEvent +=
transport.BuildCarEventHandler;      }      if
(chk_Sales.Checked)          {
pLine.Helper.BuildCarEvent +=
sales.BuildCarEventHandler;          }          // call
the main method of the subject class to do the
// actual work required, specifying any
properties required          // (in this example, just
the name of the car to build)
pLine.BuildNewMotorCar(lstCarName.Selected.Value);
// display counter values from subscribers to
show that they          // were notified by the
EventProductionLine main routine
lblResults.Text += "Cars to deliver: " +
ShowAllCarsCount(transport.CountOfCarsToDeliver);
lblResults.Text += "Cars available to sell: " +
ShowAllCarsCount(sales.CountOfCarsBuilt);
lblResults.Text += "Parts to order: " +
ShowAllCarsCount(parts.CountOfPartsUsed);      }
```



The only other relevant code in the client page is the

ShowAllCarsCount routine called by the btn_Build_Click event handler shown above. The ShowAllCarsCount routine simply iterates through the items in the Hashtable to concatenate a string that the application uses to display the car model names and counts:

```
private String ShowAllCarsCount(Hashtable
values)    {    // get list of car models and
count as a String    String result =
String.Empty;    foreach (DictionaryEntry item
in values)    {    result +=
item.Key.ToString() + ":" +
item.Value.ToString() + " ";    }    return
result + "
";    }
```

Figure 3 shows the example page after building a few cars and changing the settings in the three checkboxes so that only some subscribers receive each event.

Author's Note: You can open this page by selecting either "Publish-Subscribe" or "TransferPage4.aspx" in the drop-down list at the bottom of the Default.aspx page.

A Combined Command-Observer Pattern

A modified version of the Observer pattern can provide most of the features of the Command and the Publish-Subscribe patterns in ASP.NET. It provides for separation of the Observer and the Subject by using an interface that defines the requirements of each Observer?in much the same way as an EventHandler class defines the nature of an event in the Publish-Subscribe pattern that a subscriber can handle. A Helper class, referenced through

the Subject, maintains a list of Observers and can call the appropriate method on each one when a specified action takes place on the Subject.

In this implementation, the client or application code creates instances of the Observer objects that it wants the Subject to notify when an action occurs. For example, in the sample application, the code creates instances of a PartsDept, a TransportDept, and a SalesDept as Observer objects. It then subscribes these objects to a Helper class that maintains an internal list of subscribed Observers (see Figure 4).

The client application then calls a method on the ProductionLine class (the Subject)?an instance of which it has previously created. The ProductionLine class holds a reference to the Helper class, and calls its Notify method. The Helper class then calls the Update method on each subscribed Observer instance (see Figure 5).

Effectively, this provides a technique where the client can add commands (Observers) to a Subject without the Subject being previously aware of the nature of the target?as long as that target implements the Observer interface. At minimum, the Observer interface must expose an Update event that the Helper class can call. Depending on the nature of the implementation and requirements, the interface can specify the parameters to pass to the Observer and other methods or properties that it must expose.

Implementation of the Command-Observer Pattern

The example application demonstrates the Command-Observer pattern with an application that looks and behaves exactly the same way as the Publish-Subscribe pattern implementation shown earlier in this article. The difference is in the way that the classes

interact when the publisher/subject sends notifications to each subscriber/observer instance.

To provide independence between the observer and subject, each Observer implements an interface named `IObserver`. This allows classes to reference observer instances using the interface definition rather than the actual class type of the observer.

Alternatively, you could create a base class that contains common observer functionality, and use that rather than an interface definition.

In this simple example, the `IObserver` interface defines only a single method, named `Update`, which each observer must implement. The subject calls this method through its helper class to notify each observer whenever it builds a new car:

```
public interface IObserver {           // observed
    void Update(Object subject);        // that
                                        // updates its content or performs the
                                        // appropriate actions
}
```

As in the Publish-Subscribe example, the Command-Observer implementation uses a helper class to raise the notification to all observers. This class, named `ObserverHelper`, maintains an internal `ArrayList` that contains references to each subscribed observer. The class exposes methods for adding or removing observers from the list, using the `IObserver` interface to maintain independence between concrete classes.

The `ObserverHelper` class also exposes the `Notify` method, which the subject calls to notify all observers when the relevant action occurs in the subject class. The `Notify` method simply iterates

2/24/2022, 3:57 PM

The subject class, named `ProductionLine` in this example, carries out the main tasks of the application. It maintains a reference to an associated `ObserverHelper` instance created in its constructor and exposed through the `ObserverHelper` property?in the same way the previous Publish-Subscribe example exposed its `BuildCarHelper` class:

```
public class ProductionLine { // holds a
reference to an instance of the helper //
class that handles subscriptions and notifies all
the // subscribed observers private
ObserverHelper internalHelper = null; //
```

```

name of car currently being built      private
String carName = String.Empty;         // class
constructor      public ProductionLine(String
motorCarName)    {          // get new help class
instance         internalHelper = new
ObserverHelper();    }          // property that
exposes the helper instance      public
ObserverHelper Helper      {      get { return
internalHelper; }          }

```

In addition, like the Publish-Subscribe example, the `ProductionLine` class in this example exposes the name of the car it is currently building as a property, and a method named `BuildNewMotorCar` that saves the car model name in the local variable and calls the `Notify` method of its associated helper class:

```

// any properties required to expose data
that the      // observers may need to use.
Actual properties depend on      // actions of
subject and observer, but using properties
// maintains interface independence      public
String CurrentMotorCarName      {      get {
return carName; }          }          // main method for
class - this is called by the client      // to
carry out the main tasks. Can expose more
methods,      // but each must call the Notify
method of the associated      // helper instance
public void BuildNewMotorCar(String motorCarName)
{      // save value of car name      carName =
motorCarName;          // do whatever work
required here      // ...          // notify all

```

```

    subscribed observers
    internalHelper.Notify(this);    }    }

```

Command-Observer Pattern Implementation (continued)

The three observer-class implementations in this example, PartsDept, TransportDept, and SalesDept, maintain an internal Hashtable that holds a count of the number of each car model built, and stores it in the user's session, in exactly the same way as the Publish-Subscribe pattern example. These classes also expose this Hashtable as a property so that client applications can access it.

The main differences between the Publish-Subscribe and Command-Observer examples are that the observer classes in the Command-Observer example implement the IObserver interface, and therefore each must expose an Update method.

The Update method is the equivalent of the event handler in the Publish-Subscribe example. It casts the received object (passed to the method as a parameter instead of an event argument instance) into an instance of the ProductionLine class, and extracts any property values it requires. Then the method can update the Hashtable and store it back in the user's session. Here's the Update method implementation in the PartsDept observer class:

```

    public void Update(Object subject)    {    //
// called by ObserverHelper.Notify method. Do any
// work required in Observer class here.
// Parameter is    // the instance of the subject
// class, so code can query    // its properties
// if required    String carName = (subject as
// ProductionLine).    CurrentMotorCarName;


```

2/24/2022, 3:57 PM

Then the code calls the `BuildNewMotorCar` method, specifying the model name selected in the drop-down list in the ASPX page. This notifies each subscribed observer, and the code continues by displaying the resulting values from the Hashtable of each observer using the same `ShowAllCarsCount` method as the previous example:

```
protected void btn_Build_Click(object sender,
EventArgs e)    {           // create instance of
subject class that contains           // the method
that will notify subscribed observers
ProductionLine pLine = new ProductionLine();
// create instances of classes that will observe
```

```
the subject      //(the ProductionLine class) and
get notified when required.      // any class
that implements IObserver can be used here
PartsDept parts = new PartsDept();
TransportDept transport = new TransportDept();
SalesDept sales = new SalesDept();      //
subscribe the observers according to settings in
the page      if (chk_Parts.Checked)      {
pLine.Helper.AddObserver(parts);      }      if
(chk_Transport.Checked)      {
pLine.Helper.AddObserver(transport);      }      if
(chk_Sales.Checked)      {
pLine.Helper.AddObserver(sales);      }
```

	?
Figure 6. Command-Observer Pattern: Here's the sample application showing how it uses the Command-Observer pattern.	

```
      // call the main method of the subject
class      // to do the actual work required,
specifying      // any properties required (in
this example,      // just the name of the car to
build).
pLine.BuildNewMotorCar(lstCarName.Selected.Value);
// display counter values from observers to show
that they      // were notified by the
```

```

ProductionLine main routine      lblResults.Text
+= "Cars to deliver: "          +
ShowAllCarsCount (transport.CountOfCarsToDeliver);
lblResults.Text += "Cars available to sell: " +
ShowAllCarsCount (sales.CountOfCarsBuilt);
lblResults.Text += "Parts to order: " +
ShowAllCarsCount (parts.CountOfPartsUsed);    }

```

Figure 6 shows result of the Command-Observer pattern example after building a few cars and changing the settings in the three checkboxes so that only some observers subscribe to the notification each time.

Author's Note: You can open this page by selecting either "Command-Observer" or "TransferPage5.aspx" in the drop-down list at the bottom of the Default.aspx page.

Summing Up ASP.NET Design Patterns

From the various patterns discussed in this article series (see the Related References for links to the previous articles), it is possible to draw the following conclusions:

- The MVP and the various Controller patterns are useful, but firing update events from the Model usually is not.
- The Page Controller and Front Controller patterns allow for custom use-case behavior by showing different views or activating different presenters. Front Controller can make use of the Intercepting Filter pattern.
- The Repository pattern is useful for virtualization of source data.
- The Singleton pattern is useful for reducing the need for multiple instances.

- The Service Agent and Proxy patterns are ideal for interacting with remote services.
- The Provider pattern is useful for accessing various types of data sources.
- The Adapter pattern is useful for extending ASP.NET controls.
- The Factory, Builder, Injection, Observer, and Command patterns are less useful, more complex, and can be difficult to implement.
- Event-driven patterns such as the Publish-Subscribe pattern that rely on subscribers having a long lifetime are difficult to implement due to the nature of ASP.NET, and are probably not viable. However, the Publish-Subscribe pattern is useful if all subscribers are instantiated within the current page lifetime.
- The composite event-free Command-Observer pattern approach is useful for removing dependencies between classes.