

# The Observer Pattern

As you have been learning throughout this part of the book, design patterns are extremely useful when developing large-scale applications. Design patterns provide code flexibility and help establish a way of handling common situations that occur in the logic of an application. The Observer pattern is an extremely important addition to the sample application because it creates a way of handling an unlimited number of objects as a collection with only a few object methods. Let's learn more about how this pattern accomplishes so much with so little.

## Pattern Overview

The Observer pattern is a design pattern that is used to observe the state of an object in an application. The observed object allows other objects to register and unregister as observers. The observers are notified when specific events that they are observing are fired. The observed object has three distinct methods, which are outlined next.

### Register Observers Overview

The Observer pattern essentially allows an unlimited number of objects to observe or listen to events in the observed object (or subject) by registering themselves. After observers are registered to an event, the subject will notify them when the event is fired. The subject handles this by storing an observer collection and iterating through it when the event occurs in order to notify each observer.

### Notify Observers Overview

When an event is fired in the subject, the observers are notified via the methods they provide to the observer when they are registered. Each observer specifies its own `notify` method and defines what happens when the notification occurs. For instance, you may have an object that registers to an error-handling object and specifies a `notify` method that handles displaying the error to the user. On the other hand, you may also have a completely different object that specifies a `notify` method, that sends an email to the

developer each time an error occurs in the web application. The power of this pattern is in the fact that both of these objects can be registered and notified by the same subject.

### Unregister Observers Overview

If an object no longer wants to be notified by the subject it is registered with, it can unregister itself. There are a number of instances in which this method is useful, one of which is when you do not want an object to be notified more than one time. After an object has been notified, it can unregister itself and the subject will remove it from the observer collection. For example, imagine that you want to wait to fire a method until another method has occurred. If it follows the Observer pattern, you can register to this method and wait for notification before moving forward. Take a look at Figure 16.1 for an example.

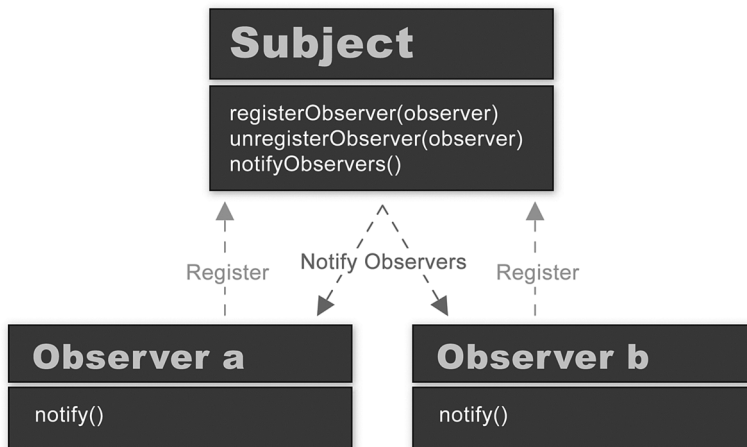


Figure 16.1 The Observer pattern allows multiple objects to observe one event and be notified when that event is invoked.

As you can see, the Observer pattern can affect numerous objects with only one event. In this chapter, we will create an error-handling object that we will call the **ErrorManager**. This object will follow the Observer pattern and notify any registered objects if an error occurs.

## Creating an Error-Handling Object

An error-handling object is essential when building and releasing large web applications that use heavy amounts of JavaScript. While building, error handling is very useful for identifying bugs in your code so that you can easily track them down and progressively finish building the application. When releasing an application, error handling is even

more important because users need to know how to handle issues that may arise and cannot do so without feedback from the application, which can tell them what went wrong. It is also important after releasing because you or your fellow developers will be able to identify what went wrong when a user is contacting you with an issue. To handle errors, let's create an object called `ErrorManager` as in Listing 16.1.

Listing 16.1 **Instantiating `ErrorManager` (`ErrorManager.js`)**

---

```
ErrorManager = {};
```

---

The core of this object is JavaScript's intrinsic `onerror` event. This method listens for errors and fires an event that can point to any custom `callback` method that you specify. The `onerror` event also has the capability to pass three parameters to the `callback` method, to provide detailed information regarding any errors that occur. The three parameters that the event passes are the actual error message, which identifies what error occurred; the URL of the document where the error occurred; and the line number in the document where the error occurred. These parameters can have a number of uses. They can be used to provide feedback to users regarding any errors that occur, or they can be used to provide feedback to the developer so that he is aware of any issues that occur during user interaction after an application has been released or is being tested. We will set this event to a local `callback` in the `ErrorManager` object, but first we need to create the observer methods.

## Register Observers

The `onerror` event will be used to notify the subject, which is the `ErrorManager` object of any errors that occur. `ErrorManager` will then notify any objects that have been registered and stored in its collection. There are three methods we will create, which we discussed in the previous section. The methods that will handle all the observation functionality are called `registerObserver`, `notifyObservers`, and `unregisterObserver`. These methods must be the first written in the object after the object has been declared. Listing 16.2 shows the `registerObserver` method.

Listing 16.2 **Registering Observers (`ErrorManager.js`)**

---

```
ErrorManager.registerObserver = function(_observer)
{
    ErrorManager.observerCollection.push(o);
}
```

---

The `registerObserver` method does exactly that: It registers an observer by adding it to an `observerCollection`, which is a property of the object and is essentially an array that is specifically used to store observers. Observers that are added to this collection need to specify the object name plus the method they want to use as the notification

callback in a *codestring*. A *codestring* is a string representation of any code. In this case, it would be a method call in the form of a string:

```
"Object.notify"
```

The collection will be declared in an `initialize` method that we will create as soon as we have finished adding the observer methods to the object so that we keep the scope of the object members intact.

## Notify Observers

The next method, called `notifyObservers` (see Listing 16.3), iterates through the `observerCollection` and fires the `callback` methods that the observers specified.

Listing 16.3 Notifying the Observers (`ErrorManager.js`)

---

```
ErrorManager.notifyObservers = function(message, url, line)
{
    for(var i in ErrorManager.observerCollection)
    {
        eval(ErrorManager.observerCollection[i] + ("'+message +'",
            "'"+ url +'','"+ line+"');");
    }
}
```

---

While iterating through the `observerCollection`, the `notifyObservers` method calls the observers by using JavaScript's intrinsic `eval` method to create a method from the *codestring* that was passed to the `registerObserver` method. The `eval` method determines whether the *codestring* is valid and executes the code if it is valid. I have also added the additional power of passing the error parameters from the `onerror` method to the `notify` methods. This allows the observers to be aware of the error that occurred and act on it as they see fit. For example, an object may register to the `ErrorManager` and receive notification when an error occurs. Based on the parameters we are passing to the `callback` method, the line of code where the error occurred may be used to highlight the corresponding issue in the GUI. This allows our objects to provide feedback to the user as to what went wrong. The best part about this pattern is that there can be an unlimited number of objects registered to the subject and they can all handle errors in different ways, depending on what portion of the application they manage. This allows for complete flexibility in your application and keeps the objects extremely decoupled, while allowing them to communicate with each other.

## Unregister Observers

The next method is the `unregisterObserver` method, which is used to remove observers from the collection so that they are no longer notified when an event occurs. Listing 16.4 shows the method as it is used in `ErrorManager`.

Listing 16.4 Unregistering Observers (`ErrorManager.js`)

---

```
ErrorManager.unregisterObserver = function(_observer)
{
    for(var observer in ErrorManager.observerCollection)
    {
        if(_observer == ErrorManager.observerCollection[observer])
        {
            ErrorManager.observerCollection.splice(observer, 1);
        }
    }
}
```

---

This method receives a codestring as did the `registerObserver` method. It then uses this string while iterating through the collection by checking to see whether there is a matching codestring in the collection. When and if there is a match, the codestring is removed from the collection. In order to remove the string from the collection, we need to use the JavaScript `splice` method to remove the specified index in the collection. After the observer has been removed from the collection, it will no longer be notified of any events that it was once registered to, although the object may register again at any time.

Once we have these three methods created, we need to initialize the object and all its properties. Listing 16.5 shows our `ErrorManager`'s `initialize` method.

Listing 16.5 Initializing the `ErrorManager` (`ErrorManager.js`)

---

```
ErrorManager.initialize = function()
{
    ErrorManager.observerCollection = new Array();
    ErrorManager.registerObserver("ErrorManager.emailError");
    onerror = ErrorManager.notifyObservers;
}
ErrorManager.initialize();
```

---

As you can see, the `initialize` method is very important to this object. It must be called as soon as it has been created to allow proper scoping of its properties. As I mentioned earlier, it handles the creation of the `observerCollection` that we used throughout all of the observer methods. The `observerCollection` is nothing more than a simple array that is used to collect the observers as they are registered. The next two lines of code in this method register two methods from the `ErrorManager`: a

method named `emailError` and another named `alert`. These two methods will be notified when any JavaScript error occurs in the application. Now that we have the observer methods created and the objects' properties are available, we can set the local `callback` method to JavaScript's intrinsic `onerror` event. This event becomes the core of this object; without it, the object would not function. Anytime a JavaScript error occurs, this event fires the callback, which in this case is the `notifyObservers` method. It may look as if we are defining a variable, but since this is an intrinsic JavaScript event it knows how to handle the assignment. This method then handles notifying all the observers of the error that occurred, along with all the details, including the message, URL, and exact line of code.

The first notify method that we will create is the `emailError` method (see Listing 16.6) that we registered in the `initialize` method. This method contains the functionality to send an email to specified developers listing all the details of the error.

Listing 16.6 **Emailing Errors** (`ErrorManager.js`)

---

```

EventManager.emailError = function(message, url, line)
{
    var error = "<b>Error:</b> <font color=red>"+ message + "</font><br/>";
    error += "<b>URL:</b> " + url + "<br/>";
    error += "<b>Line:</b> " + line + "<br/>";

    var page = "classes/EventManager.class.php";
    var subject = "My Ajax Application Error";
    AjaxUpdater.Update('POST', page + "?subject="+ subject + "&message="+ error,
        ↪this.catchResponse);
}

EventManager.catchResponse = function()
{
    if(Ajax.checkReadyState('loading') == "OK")
    {
        // Handle the response from the server-side
    }
}

```

---

Take a look at the parameters the `emailError` method accepts. They are the three parameters that we previously discussed. We will use these parameters to create an HTML-formatted email that we will ultimately send to ourselves or the developer of our application. After we have the error formatted as HTML, we will identify the variables that will be used as the parameters in the email. The first parameter is the page that we are requesting through Ajax, which is an `ErrorManager` class that we will create with PHP shortly, and the second is the subject of the email. After we have our variables identified for the request, we will send it to the `AjaxUpdater`'s `update` method, which we created in Chapter 14, "Singleton Pattern." This request will be made through the

POST method to the page that we specify, which in this case is an `ErrorManager` class that we will create with PHP. The request is sent along with a query string that consists of the variables we defined, such as the subject and HTML error. We will also specify a callback method, named `catchResponse`, which we can use to handle the response from the server. After all this information has been passed to the `AjaxUpdater`, it will make the request through the Ajax object to the PHP page specified.

Although we have not covered server-side interaction with Ajax, in this section we will be jumping ahead a bit in order to add the functionality to send the email. Later, in Part V, “Server-Side Interaction,” we will take an in-depth look at server-side interaction. The PHP `ErrorManager` class is a fairly small class, so it will be a good object in which to make our first server-side interaction. Listing 16.7 shows the class in all its glory.

### Note

The PHP classes used throughout this book require PHP version 5.0.

Listing 16.7 `ErrorManager` PHP Class (`ErrorManager.class.php`)

```
<?php

$errorManager = new ErrorManager();
$errorManager->send($_GET["subject"], $_GET["message"]);

class ErrorManager
{

    public function ErrorManager() {}

    public function send($_subject, $_message)
    {
        $headers = "From: noreply@sample.com\r\n" .
            "Reply-To: noreply@sample.com\r\n" .
            "MIME-Version: 1.0\r\n" .
            "Content-Type: text/html; charset=utf-8\r\n" .
            "Content-Transfer-Encoding: 8bit\r\n\r\n";
        mail("you@yourdomain.com", $_subject, $_message, $headers);
    }

}

?>
```

As I mentioned, this class is fairly small because it only consists of one method. It is important to notice that we are instantiating the object in the same file. This allows us to make requests directly to the class without having to create an intermediate file that does

it for us. Either way is perfectly fine, but I find this way to be a bit cleaner and easier to manage. After the object is instantiated, we call the `send` method and pass it two parameters, which are the `subject` and `message` parameters. You may recognize these parameters from the `ErrorManager` object that we created in JavaScript. These are the `onerror` parameters that are sent to the `callback` method and ultimately passed through the request to this class. Therefore, this method is invoked as soon as the page is requested. The request is then handled by the `send` method, which sends an email based on the parameters that are sent to it. However, before it sends the email, it also specifies the headers, such as the from email address, the reply-to email address, and the MIME version, content type, and content-transfer encoding, which are all used to handle the HTML format that we are passing through the request. After the headers have been identified, we will use PHP's intrinsic mail function, which allows you to send an email.

The only requirement is that the server you are using to run the web application must have access to a mail application. In order to get more information about the mail function, you may visit the PHP manual at [www.php.net](http://www.php.net) and search for *mail* in the function list. This page will provide you with additional information regarding the function and samples to show you how to make our `ErrorManager` class more robust. For example, you may want to add error handling in the `ErrorManager` class in case the email is not sent for any reason. If an error occurs, you can then respond to the client-side Ajax engine and ultimately the `catchResponse` method in the JavaScript `ErrorManager` with the response. The JavaScript `ErrorManager` can then handle the response by trying to send another email if there was a failure or simply notifying the user of the error that occurred as an alternative to sending the email.

Now that we have a fully functional `ErrorManager` object, we can use it throughout our application. Let's see how.

## Using the Error-Handling Object

Error handling is essential to a successful web application, but it is often overlooked and deemed a boring chore to developers. Using the Observer pattern for Ajax error handling allows us to create an extremely intuitive user experience and is actually quite interesting to program. As you learned in Chapter 9, "Extending the Engine," with HTTP status codes, we can also provide immediate feedback and options to the user when errors occur without disrupting the flow of experience or leaving a user with questions regarding what to do next. Imagine having an object that can control all the alerts in your application. I have created one that is named `Alert`. This object registers an error method with the `ErrorManager` object immediately after it is instantiated. This method accepts the three parameters the `notifyObservers` method passes to observers in its collection. It then concatenates an error string that is displayed in an alert box to the user (see Listing 16.8).

---

**Listing 16.8 Alert Object (Alert.js)**

---

```
Alert = {};  
  
ErrorManager.registerObserver("Alert.error");  
  
Alert.error = function(message, url, line)  
{  
    var error = "Error: "+ message +"\n";  
    error += "URL: "+ url +"\n";  
    error += "Line: "+ line;  
    alert(error);  
    //ErrorManager.unregisterObserver("Alert.error");  
}
```

---

After the error method is called, you can unregister it as an observer by uncommenting the last line in the method. This is only one simple example of how to use the Observer pattern in a web application. There are millions of ways this pattern can bring power to your applications.