Submitted by chco
Posted Nov 02 2010 02:11 PM

This excerpt from Javascript Patterns (http://oreilly.com/catalog/9780596806767?cmp=il-orm-ans-learnmore-9780596806767) explains the Javascript observer pattern where a subject object automatically notifies its observer objects of any state changes.The *observer* pattern is widely used in client-side Javascript programming. All the browser events (mouseover, keypress, and so on) are examples of the pattern. Another name for it is also *custom events*, meaning events that you create programmatically, as opposed to the ones that the browser fires. Yet another name is *subscriber/publisher* pattern.

The main motivation behind this pattern is to promote loose coupling. Instead of one object calling another object's method, an object subscribes to another object's specific activity and gets notified. The subscriber is also called observer, while the object being observed is called publisher or subject. The publisher notifies (calls) all the subscribers when an important event occurs and may often pass a message in the form of an event object.

Example #1: Magazine Subscriptions

To understand how to implement this pattern, let's take a concrete example. Let's say you have a publisher `paper`, which publishes a daily newspaper and a monthly magazine. A subscriber `joe` will be notified whenever that happens.

The `paper` object needs to have a property `subscribers` that is an array storing all subscribers. The act of subscription is merely adding to this array. When an event occurs, `paper` loops through the list of subscribers and notifies them. The notification means calling a method of the subscriber object. Therefore, when subscribing, the subscriber provides one of its methods to `paper's subscribe()` method.

The `paper` can also provide `unsubscribe()`, which means removing from the array of subscribers. The last important method of `paper` is `publish()`, which will call the subscribers' methods. To summarize, a publisher object needs to have these members:

`subscribers`

An array

`subscribe()`

Add to the array of subscribers

`unsubscribe()`

Remove from the subscribers array

`publish()`

Loop though subscribers and call the methods they provided when they signed up

All the three methods need a `type` parameter, because a publisher may fire several events (publish both a magazine and a newspaper) and subscribers may chose to subscribe to one, but not to the other.

Because these members are generic for any publisher object, it makes sense to implement them as part of a separate object. Then we can copy them over (mix-in pattern) to any object and turn any given object into a publisher.

Here's an example implementation of the generic publisher functionality, which defines all the required members previously listed plus a helper `visitSubscribers()` method:

```
var publisher = {
   subscribers: {
      any: [] // event type: subscribers
   },
   subscribe: function (fn, type) {
      type = type || 'any';
      if (typeof this.subscribers[type] === "undefined") {
         this.subscribers[type] = [];
      }
      this.subscribers[type].push(fn);
   },
   unsubscribe: function (fn, type) {
      this.visitSubscribers('unsubscribe', fn, type);
   },
   publish: function (publication, type) {
      this.visitSubscribers('publish', publication, type);
   },
   visitSubscribers: function (action, arg, type) {
      var pubtype = type || 'any',
         subscribers = this.subscribers[pubtype],
         i,
         max = subscribers.length;

      for (i = 0; i < max; i += 1) {
         if (action === 'publish') {
            subscribers[i](arg);
         } else {
            if (subscribers[i] === arg) {
               subscribers.splice(i, 1);
            }
         }
      }
   }
};
```

And here's a function that takes an object and turns it into a publisher by simply copying over the generic publisher's methods:

```
function makePublisher(o) {
   var i;
```

```
    for (i in publisher) {
        if (publisher.hasOwnProperty(i) && typeof publisher[i] === "function") {
            o[i] = publisher[i];
        }
    }
    o.subscribers = {any: []};
}
```

Now let's implement the `paper` object. All it can do is publish daily and monthly:

```
var paper = {
    daily: function () {
        this.publish("big news today");
    },
    monthly: function () {
        this.publish("interesting analysis", "monthly");
    }
};
```

Making paper a publisher:

```
makePublisher(paper);
```

Now that we have a publisher, let's see the subscriber object `joe`, which has two methods:

```
var joe = {
    drinkCoffee: function (paper) {
        console.log('Just read ' + paper);
    },
    sundayPreNap: function (monthly) {
        console.log('About to fall asleep reading this ' + monthly);
    }
};
```

Now the paper subscribes `joe` (in other words `joe` subscribes to the `paper`):

```
paper.subscribe(joe.drinkCoffee);
paper.subscribe(joe.sundayPreNap, 'monthly');
```

As you see, joe provides a method to be called for the default “any” event and another method to be called when the “monthly” type of event occurs. Now let's fire some events:

```
paper.daily();
paper.daily();
paper.daily();
paper.monthly();
```

All these publications call `joe`'s appropriate methods and the result in the console is:

```
Just read big news today
Just read big news today
Just read big news today
About to fall asleep reading this interesting analysis
```

The good part here is that the `paper` object doesn't hardcode `joe` and `joe` doesn't hardcode `paper`. There's also no mediator object that knows everything. The participating objects are loosely coupled, and without modifying them at all, we can add many more subscribers to `paper`; also `joe` can unsubscribe at any time.

Let's take this example a step further and also make `joe` a publisher. (After all, with blogs and microblogs anyone can be a publisher.) So `joe` becomes a publisher and can post status updates on Twitter:

```
makePublisher(joe);
joe.tweet = function (msg) {
    this.publish(msg);
};
```

Now imagine that the paper's public relations department decides to read what its readers tweet and subscribes to joe, providing the method `readTweets()`:

```
paper.readTweets = function (tweet) {
    alert('Call big meeting! Someone ' + tweet);
};
joe.subscribe(paper.readTweets);
```

Now as soon as `joe` tweets, the paper is alerted:

```
joe.tweet("hated the paper today");
```

The result is an alert: “Call big meeting! Someone hated the paper today.”

You can see the full source code and play in the console with a live demo at http://jspatterns.co...7/observer.html (http://jspatterns.com/book/7/observer.html) .

Example #2: The Keypress Game

Let’s take another example. We’ll implement the same keypress game from the mediator pattern example but this time using the observer pattern. To make it slightly more advanced, let’s accept an unlimited number of players, not only two. We’ll still have the `Player()` constructor that creates player objects and the `scoreboard` object. Only the `mediator` will now become a `game` object.

In the mediator pattern the `mediator` object knows about all other participating objects and calls their methods. The `game` object in the observer pattern will not do that; instead it will leave it to the objects to subscribe to interesting events. For example, the `scoreboard` will subscribe to `game`’s “scorechange” event.

Let’s first revisit the generic `publisher` object and tweak its interface a bit to make it closer to the browser world:

- Instead of publish(), subscribe(), and unsubscribe(), we’ll have fire(), on(), and remove().
- The type of event will be used all the time, so it becomes the first argument to the three functions.
- An extra context can be supplied in addition to the subscriber's function to allow the callback method to use this referring to its own object.

The new `publisher` object becomes:

```
var publisher = {
   subscribers: {
      any: []
   },
   on: function (type, fn, context) {
      type = type || 'any';
      fn = typeof fn === "function" ? fn : context[fn];

      if (typeof this.subscribers[type] === "undefined") {
         this.subscribers[type] = [];
      }
      this.subscribers[type].push({fn: fn, context: context || this});
   },
   remove: function (type, fn, context) {
      this.visitSubscribers('unsubscribe', type, fn, context);
   },
   fire: function (type, publication) {
      this.visitSubscribers('publish', type, publication);
   },
   visitSubscribers: function (action, type, arg, context) {
      var pubtype = type || 'any',
         subscribers = this.subscribers[pubtype],
         i,
         max = subscribers ? subscribers.length : 0;

      for (i = 0; i < max; i += 1) {
         if (action === 'publish') {
            subscribers[i].fn.call(subscribers[i].context, arg);
         } else {
            if (subscribers[i].fn === arg && subscribers[i].context === context) {
               subscribers.splice(i, 1);
            }
         }
      }
   }
};
```

The new `Player()` constructor becomes:

```
function Player(name, key) {
   this.points = 0;
   this.name = name;
   this.key  = key;
   this.fire('newplayer', this);
}

Player.prototype.play = function () {
   this.points += 1;
   this.fire('play', this);
};
```

The new parts here are that the constructor accepts key, the keyboard key that the player presses to make points. (The keys were hardcoded before.) Also, every time a new player object is created, an event “newplayer” is fired. Similarly, every time a player plays, the event “play” is fired.

The `scoreboard` object remains the same; it simply updates the display with the current score.

The new game object can keep track of all players, so it can produce a score and fire a “scorechange” event. It will also subscribe to all “keypress” events from the browser and will know about the keys that correspond to each player:

```
var game = {

   keys: {},

   addPlayer: function (player) {
      var key = player.key.toString().charCodeAt(0);
      this.keys[key] = player;
   },

   handleKeypress: function (e) {
        e = e || window.event; // IE
        if (game.keys[e.which]) {
            game.keys[e.which].play();
        }
    },

    handlePlay: function (player) {
        var i,
            players = this.keys,
            score = {};

        for (i in players) {
            if (players.hasOwnProperty(i)) {
                score[players[i].name] = players[i].points;
            }
        }
        this.fire('scorechange', score);
    }
};
```

The function `makePublisher()` that turns any object into a publisher is still the same as in the newspaper example. The `game` object becomes a publisher (so it can fire "scorechange" events) and the `Player.protoype` becomes a publisher so that every player object can fire "play" and "newplayer" events to whomever decides to listen:

```
makePublisher(Player.prototype);
makePublisher(game);
```

The `game` object subscribes to "play" and "newplayer" events (and also "keypress" from the browser), while the `scoreboard` subscribes to "scorechange":

```
Player.prototype.on("newplayer", "addPlayer",  game);
Player.prototype.on("play",      "handlePlay", game);
game.on("scorechange", scoreboard.update, scoreboard);
window.onkeypress = game.handleKeypress;
```

As you see here, the `on()` method enables subscribers to specify the callback as a reference to a function (`scoreboard.update`) or as a string (`"addPlayer"`). The string works as long as the context (for example, `game`) is also provided.

The final bit of setup is to dynamically create player objects (with their corresponding keys to be pressed), as many as the user wants:

```
var playername, key;
while (1) {
    playername = prompt("Add player (name)");
    if (!playername) {
        break;
    }
    while (1) {
        key = prompt("Key for " + playername + "?");
        if (key) {
            break;
        }
    }
    new Player(playername,  key);
}
```

And that concludes the game. You can see the full source and play it at http://jspatterns.co...erver-game.html (http://jspatterns.com/book/7/observer-game.html) .

Notice that in the mediator pattern implementation, the `mediator` object had to know about every other object to call the correct methods at the right time and coordinate the whole game. Here the `game` object is a little dumber and relies on the objects observing certain events and taking action: for example, the `scoreboard` listening to the "scorechange" event. This results in an even looser coupling (the less one object knows, the better) at the price of making it a little harder to keep track of who listens to what event. In this example game, all the subscriptions happened in the same place in the code, but as an application grows, the `on()` calls may be all over the place (for example, in each object's initialization code). This will make it harder to debug, as there will be no single place to look at the code and understand what's going on. In the observer pattern, you move away from the procedural sequential code execution where you start from the beginning and can follow the program to the end.

0 Replies

Reply