عنوان: **الگوی مشاهده‌گر Observer Pattern**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۵/۰۱/۱۴ ۲۰:۱۵
آدرس: www.dotnettips.info
گروه‌ها: Design patterns

الگوی مشاهده‌گر یکی از محبوبترین و معروفترین الگوهای برنامه نویسی است که پیاده سازی آن در بسیاری از زبان‌ها رواج یافته است. برای نمونه پیاده سازی این الگو را می‌توانید در بسیاری از کتابخانه‌ها (به خصوص GUI) مانند این مطالب (+ + +) مشاهده کنید. برای اینکه بتوانیم این الگو را خودمان برای اشیاء برنامه خودمان پیاده کنیم، بهتر است که بیشتر با خود این الگو آشنا شویم. برای شروع بهتر است که با یک مثال به تعریف این الگو بپردازیم. مثال زیر نقل قولی از یکی از مطالب این سایت است که به خوبی کارکرد این الگو را برای شما نشان میدهد.

یک لامپ و سوئیچ برق را درنظر بگیرید. زمانیکه لامپ مشاهده می‌کند سوئیچ برق در حالت روشن قرار گرفته‌است، روشن خواهد شد و برعکس. در اینجا به سوئیچ، subject و به لامپ، observer گفته می‌شود. هر زمان که حالت سوئیچ تغییر می‌کند، از طریق یک callback، وضعیت خود را به observer اعلام خواهد کرد. علت استفاده از callbackها، ارائه راه‌حل‌های عمومی است تا بتواند با انواع و اقسام اشیاء کار کند. به این ترتیب هر بار که شیء observer از نوع متفاوتی تعریف می‌شود (مثلا بجای لامپ یک خودرو قرار گیرد)، نیازی نخواهد بود تا subject را تغییر داد.

عموما به شیءایی که قرار است وضعیت را مشاهده یا رصد کند، Observer گفته می‌شود و به شیءایی که قرار است وضعیت آن رصد شود Observable یا Subject گفته می‌شود.
بد نیست بدانید این الگو یکی از کلیدی‌ترین بخش‌های معماری لایه بندی MVC نیز می‌باشد.
همچنین این نکته حائز اهمیت است که این الگو ممکن است باعث نشتی حافظه هم شود و به این مشکل Lapsed Listener Problem می‌گویند. یعنی یک listener وجود دارد که تاریخ آن منقضی شده، ولی هنوز در حافظه جا خوش کرده‌است. این مشکل برای زبان‌های شیءگرایی که با سیستمی مشابه GC پیاده سازی می‌شوند، رخ میدهد. برای جلوگیری از این حالت، برنامه نویس باید این مشکل را با رجیستر کردن‌ها و عدم رجیستر یک شنونده، در مواقع لزوم حل کند. در غیر این صورت این شنونده بی جهت، یک ارتباط را زنده نگه داشته و حافظه‌ی منبع را به هدر میدهد.

مثال: ما یک کلید داریم که سه کلاس RedLED،GreenLED و BlueLED قرار است آن را مشاهده و وضعیت کلید را رصد کنند.
برای پیاده سازی این الگو، ابتدا یک کلاس انتزاعی را با نام Observer که دارای متدی به نام Update است، ایجاد می‌کنیم. متغیر از نوع کلاس Observable را بعدا ایجاد می‌کنیم:

```
public abstract class Observer
    {
        protected Observable Observable;
        public abstract void Update();
    }
```

سپس یک کلاس پدر را به نام Observable می‌سازیم تا آن را به شیء سوئیچ نسبت دهیم:

```
public  class Observable
    {
        private readonly List<Observer> _observers = new List<Observer>();

        public void Attach(Observer observer)
        {
            _observers.Add(observer);
        }
        public void Dettach(Observer observer)
        {
            _observers.Remove(observer);
        }

        public void NotifyAllObservers()
        {
```

```
            foreach (var observer in _observers)
            {
                observer.Update();
            }
        }
    }
```

در کلاس بالا یک لیست از نوع Observerها داریم که در آن، کلید با تغییر وضعیت خود، لیست رصد کنندگانش را مطلع می‌سازد و دیگر چراغ‌های LED نیازی نیست تا مرتب وضعیت کلید را چک کنند. متدهای attach و Detach در واقع همان رجیسترها هستند که باید مدیریت خوبی روی آن‌ها داشته باشید تا نشتی حافظه پیش نیاید. در نهایت متد NotifyAllObservers هم متدی است که با مرور لیست رصدکنندگانش، رویداد Update آن‌ها را صدا میزند تا تغییر وضعیت کلید به آن‌ها گزارش داده شود.

حال کلاس Switch را با ارث بری از کلاس Observable می‌نویسیم:

```
public  class Switch:Observable
    {
        private bool _state;

        public bool ChangeState
        {
            set
            {
                _state = value;
                NotifyAllObservers();
            }
            get { return _state; }
        }
    }
```

در کلاس بالا هرجایی که وضعیت کلید تغییر می‌یابد، متد NotifyAllObservers صدا زده میشود.
برای هر سه چراغ، رنگی هم داریم:

```
public class RedLED:Observer
    {
        private bool _on = false;
        public override void Update()
        {
            _on = !_on;
            Console.WriteLine($"Red LED is {((_on) ? "On" : "Off")}");
        }
    }
```

```
public class GreenLED:Observer
    {
        private bool _on = false;
        public override void Update()
        {
            _on = !_on;
            Console.WriteLine($"Green LED is {((_on) ? "On" : "Off")}");
        }
    }
```

```
public  class BlueLED:Observer
    {
        private bool _on = false;
        public override void Update()
        {
            _on = !_on;
            Console.WriteLine($"Blue LED is {((_on) ? "On" : "Off")}");
        }
    }
```

سپس در Main اینگونه می‌نویسیم:

```
var greenLed=new GreenLED();
            var redLed=new RedLED();
            var blueLed=new BlueLED();

            var switchKey=new Switch();
            switchKey.Attach(greenLed);
            switchKey.Attach(redLed);
            switchKey.Attach(blueLed);

            switchKey.ChangeState = true;
            switchKey.ChangeState = false;
```

به طور خلاصه هر سه چراغ به شیء کلید attach شده و با هر بار عوض شدن وضعیت کلید، متدهای Update هر سه چراغ صدا زده خواهند شد. نتیجه‌ی کد بالا به شکل زیر در کنسول نمایش می‌یابد:

```
Green LED is On
Red LED is On
Blue LED is On
Green LED is Off
Red LED is Off
Blue LED is Off
```

نظرات خوانندگان

نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۵/۰۴/۲۱ ۱۷:۲۴

یکی از مثال هایی که خودم چند روز پیش از این الگو در جاوا برای اندروید استفاده کردم این است که در بخشی از برنامه انتقال اطلاعاتی صورت میگرفت که شاید نیاز باشد هزاران آیتم در برنامه انتقال یابند که در این صورت بهتر بود که فعالیت توسط یک نوار پیشرفت نمایش داده شود و از آنجا که این انتقال برای آیتم‌ها در کلاسی جداگانه قرار داشت و در این حالت ممکن بود تکه کد نامربوط بین بخش رابط کاربری و منطق اضافه کند و وابستگی ایجاد کند از این الگو استفاده کردم. در این حالت هر وقت متد مورد نظر انتقالی انجام میداد کلاس پیشرفت را برای محاسبه درصد آگاه می‌ساخت.