عنوان: **الگوی بازدیدکننده Visitor Pattern**
نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۵/۰۱/۲۶ ۰:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns

این الگو یکی دیگر از الگوهای رفتاری است که به قاعده OCP یا Open Closed Principle کمک بسیاری می‌کند. این الگو برای زمانی مناسب است که ما سعی بر این داریم تا یک سری الگوریتم‌های متفاوت را بر روی یک سری از اشیاء پیاده سازی کنیم. به عنوان مثال تصور کنید که ما در یک سازمان افراد مختلفی را از مدیریت اصلی گرفته، تا ساده‌ترین کارمندان، داریم و برای محاسبه حقوق و مالیات و ... نیاز است تا برای هر کدام دستور العمل‌هایی را اجرا کنیم و ممکن است در آینده تعداد این دستور العمل‌ها بالاتر هم برود.
در این مثال ما سه گروه Manager,Employee و Worker را داریم که می‌خواهیم با استفاده از این الگو برای هر کدام به طور جداگانه، حقوق و دستمزد و اضافه کاری را محاسبه کنیم. با توجه به اینکه فرمول هر یک جداست و این احتمال نیز وجود دارد که هر کدام خواص مخصوص به خود را داشته باشند که در دیگری وجود ندارد و در آینده این احتمال می‌رود که سمت جدید یا دستورالعمل‌های جدیدی اضافه شود، بهترین راه حل استفاده از الگوی Visitor است.

الگوی visitor دو بخش مهم دارد؛ یکی Element که قرار است کار روی آن انجام شود. مثل سمت‌های مختلف و دیگری Visitor هست که همان دستورالعمل‌هایی چون محاسبه حقوق و دستمزد و ... است که روی المان‌ها صورت می‌گیرد.
ابتدا برای هر کدام یک اینترفیس را با مشخصات زیر می‌سازیم:

```
public interface IElement
    {
        void Accept(IElementVisitor visitor);
    }
```

```
public interface IElementVisitor
    {
        void Visit(Manager manager);
        void Visit(Employee manager);
        void Visit(Worker manager);
    }
```

همانطور که می‌بینید در کلاس Visitor سه متد هستند که سه کلاس مدیر، کارمند و کارگر را که مشتق شده از اینترفیس Element هستند، به صورت آرگومان می‌پذیرند. توصیف هر کلاس المان به شرح زیر است:

```
public class  Manager: IElement
    {
        public int WorkingHour = 8;
        public int Wife = 1;
        public int Children = 3;
        public int OffDays = 6;
        public int OverHours = 12;

        public void Accept(IElementVisitor visitor)
        {
            visitor.Visit(this);
        }
    }
```

```
public class Employee: IElement
    {
        public int WorkingHour = 8;
        public int Wife = 1;
        public int Children = 3;
        public int OffDays = 6;
        public int OverHours = 12;

        public void Accept(IElementVisitor visitor)
```

```
        {
            visitor.Visit(this);
        }
    }
```

```
public class Worker:IElement
    {
        public int WorkingHour = 8;
        public int Wife = 1;
        public int Children = 3;
        public int OffDays = 6;
        public int OverHours = 12;

        public void Accept(IElementVisitor visitor)
        {
            visitor.Visit(this);
        }
    }
```

ما اطلاعات هر کلاس را در این مثال، مشابه گذاشته‌ایم تا نتیجه فرمول را ببینیم. ولی هیچ الزامی به رعایت آن نیست.
حال وقت آن رسیده تا از روی کلاس Visitor، برای حقوق، دستمزد و اضافه کاری، کلاس‌های جدیدی را بسازیم:

```
class SalaryCalculator:IElementVisitor
    {
        public void Visit(Manager manager)
        {
            var salary = manager.WorkingHour*10000;
            salary += manager.Wife*25000;
            salary += manager.Children*20000;
            salary -= manager.OffDays*5000;
            Console.WriteLine("Manager's Salary is " + salary);
        }

        public void Visit(Employee employee)
        {
            var salary = employee.WorkingHour * 7000;
            salary += employee.Wife * 15000;
            salary += employee.Children * 10000;
            salary -= employee.OffDays * 6000;
            Console.WriteLine("Employee's Salary is " + salary);
        }

        public void Visit(Worker worker)
        {
            var salary = worker.WorkingHour * 6000;
            salary += worker.Wife * 5000;
            salary += worker.Children * 2000;
            salary -= worker.OffDays * 7000;
            Console.WriteLine("Worker's Salary is " + salary);
        }
    }
```

```
class WageCalculator:IElementVisitor
    {
        public void Visit(Manager manager)
        {
            var wage = manager.OverHours*30000;
            Console.WriteLine("Employee's wage is " + wage);
        }

        public void Visit(Employee employee)
        {
            var wage = employee.OverHours * 20000;
            Console.WriteLine("Employee's wage is " + wage);
        }

        public void Visit(Worker worker)
        {
            var wage = worker.OverHours * 15000;
            Console.WriteLine("Employee's wage is " + wage);
        }
```

```
    }
```

اکنون نیاز است تا ارتباط بین المان‌ها و بازدید کننده‌ها را طوری برقرار کنیم که برای تغییر آن‌ها در آینده، مشکلی نداشته باشیم. به همین جهت یک کلاس جدید به نام سیستم مالی ایجاد می‌کنیم:

```
class FinancialSystem
    {
        private readonly IList<IElement> _elements;

        public FinanceSystem()
        {
            _elements=new List<IElement>();
        }

        public void Attach(IElement element)
        {
            _elements.Add(element);
        }

        public void Detach(IElement element)
        {
            _elements.Remove(element);
        }

        public void Accept(IElementVisitor visitor)
        {
            foreach (var element in _elements)
            {
                element.Accept(visitor);
            }
        }
    }
```

در این روش تمام المان‌ها را داخل یک لیست قرار داده و سپس با استفاده از متد Accept، یکی از کلاس‌های مشتق شده از Visitor را به آن نسبت می‌دهیم که وظیفه آن صدا زدن متد Accept درون المان هاست. وقتی متد Accept المان‌ها صدا زده شد، شیء، المان را به متد Visit در Visitor داده و فرمول را روی آن اجرا می‌کند.
بدنه اصلی:

```
IElement manager=new Manager();
IElement employee=new Employee();
IElement worker=new Worker();

var fine=new FinancialSystem();
fine.Attach(manager);
fine.Attach(employee);
fine.Attach(worker);

fine.Accept(new SalaryCalculator());
fine.Accept(new WageCalculator());
```

نتیجه خروجی:

```
Manager's Salary is 135000
Employee's Salary is 65000
Worker's Salary is 17000
Manager's wage is 360000
Employee's wage is 240000
Worker's wage is 180000
```