

ASP.NET MVC и углубление в изучение C#

Введение в паттерны. Структурные



На этом уроке

1. Познакомимся с очередной группой паттернов.
2. Поймём, для чего они используются.
3. Рассмотрим разницу между ними и то, как их называют.
4. Разберём такие паттерны, как Facade, Adapter и Decorator.
5. Увидим разницу между данными паттернами.

Оглавление

[Введение в паттерны 2](#)

[Группа структурные паттерны](#)

[Decorator](#)

[Façade](#)

[Adapter](#)

[Вывод](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Введение в паттерны 2

ООП является мощным инструментом в разработке программного обеспечения. Благодаря объектно-ориентированному подходу можно создавать гибкий код, который может с легкостью изменяться под тяжелым прессом новых и новых требований от заказчика или аналитика. Все же, надеюсь, понимают, что сложно предугадать все сразу, и правок, к сожалению, не избежать даже на последней стадии разработки программного обеспечения.

Но вот представьте, вам дают задачу, которая ведет за собой изменение поведения какого-либо объекта, который не может быть унаследован, но новый функционал должен быть именно в нем. Изменение класса такого объекта может повлиять на всю систему в целом и может привести к непредсказуемым результатам – ошибкам, особенно в бизнес-логике. Это очень высокий риск, на которой может пойти уж очень храбрый разработчик. Но как известно – проявление храбрости не всегда является хорошим решением.

Так, что же делать в данной ситуации? Требуется новый функционал, а изменения подвержены риску ошибок старой логики? На этот вопрос и на многие другие вопросы в этом же духе можно ответить так – использовать структурные паттерны. Это безопасный подход для добавления динамического функционала, агрегации старого неудобного функционала в один, приведение одного объекта под другой интерфейс.

Группа структурные паттерны

Данная группа паттернов часто применяется, как и в новом программном обеспечении так и в старом – унаследованном коде. Данные паттерны позволяют изменять и дополнять код, так, что в последствии можно избежать каких-либо эффектов на старой логике, как было обозначено ранее. Одни из самых часто используемых паттернов данной группы – фасад, адаптер и декоратор. И именно о них и их разницы, которую зачастую не видят новые разработчики, пойдет данный урок.

Decorator

Давайте представим очередной гипотетический пример. Есть некий класс, который уже используется в миллионах других классов, в модульных тестах и в целом скрыт от вас пометкой `sealed`, и находится в закрытой сборке, к коду которой у вас нет доступа. Это довольно типичная проблема, особенно когда используете сторонний пакет из `nuget` сообщества. Этот класс калькулирует, допустим, некую сумму

продаж от начала и конца конкретного периода. Приведем упрощенное представление данного класса.

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;

namespace Patterns.Factories
{
    public sealed class Order
    {
        public int Id { get; set; }

        public DateTime CreationDateTime { get; set; }

        public long TotalMoney { get; set; }
    }

    public interface IOrdersRepository
    {
        IReadOnlyList<Order> GetAll();
    }

    public interface ICalculation
    {
        long CalculateByDate(DateTime from, DateTime to);
    }

    public sealed class Calculation : ICalculation
    {
        private readonly IOrdersRepository _ordersRepository;

        public Calculation(IOrdersRepository ordersRepository)
        {
            _ordersRepository = ordersRepository;
        }

        public long CalculateByPeriod(DateTime from, DateTime to)
        {
            long total =
                _ordersRepository
                    // Получаем все заказы
                    .GetAll()
                    //Фильтруем по периоду
                    .Where(x => x.CreationDateTime >= from &&
x.CreationDateTime <= to)
                    //Суммируем сумму
                    .Sum(x => x.TotalMoney);

            return total;
        }
    }
}
```

Вам дали задачу сделать новый функционал, который должен, к примеру принимать массив периодов и выдавать списком подсчет. Логически новый метод идеально вписался бы в существующий класс, но расширять у нас возможности нет, так как он находится в закрытой сборке, а создавать новый – неправильно. И именно на такие случаи идеально подходит паттерн Decorator, который расширяет данный класс на дополнительный метод.

```
public sealed class Period
{
    public DateTime From { get; set; }
    public DateTime To { get; set; }
    public long Total { get; internal set; }
}

public sealed class AdvancedCalculation : ICalculation
{
    private readonly Calculation _calculation;

    public AdvancedCalculation(Calculation calculation)
    {
        _calculation = calculation;
    }

    public long CalculateByDate(DateTime from, DateTime to)
    {
        if (_calculation is null)
        {
            return 0;
        }

        return _calculation.CalculateByDate(from, to);
    }

    public Period CalculateByDate(Period period)
    {
        if (_calculation is null || period is null)
        {
            return period;
        }

        period.Total = CalculateByDate(period.From, period.To);

        return period;
    }

    public IEnumerable<Period> CalculateByPeriods(IEnumerable<Period>
periods)
    {
        if (periods is null)
        {
            yield break;
        }

        foreach (Period period in periods)
        {
            //Постепенно возвращаем уже подсчитанный период на большом
множестве
            yield return CalculateByDate(period);
        }
    }
}
```

```
}  
}  
}
```

В итоге старый класс не подвергся никаким изменениям, что приводит к тому, что старая логика не изменилась, а просто задекорировалась. А вот новая логика уже будет использовать новый класс, с новыми возможностями. В данном примере стоит обратить внимание, что есть общий интерфейс `ICalculation`, в котором наследуется и старый и новый класс.

Façade

Вы пришли в офис, открыли доску с задачами и натываетесь на задачу, которая выглядит достаточно просто. Нужно взять какой-либо сервис, потом еще сервис, потому что в этих сервисах лежит нужная логика и открыт доступ к нужным методам. И вот уже задача не кажется простой. Зависимостей становится все больше и больше, и больше. Конечный потребитель данных сервисов становится уж очень громоздким и неповоротливым. Но не унывайте! Вам придет на помощь паттерн `Facade`, который скроет сложную логику для конечного потребителя.

Опять же разберем гипотетический пример. Надо сделать универсальный клиент для формирования отчетности. Под отчетностью подразумевается получение и списка заказов, и пользователей и что-то еще, что может потребоваться для отчетной системы.

```
using System;

using System.Collections.Concurrent;

using System.Collections.Generic;

using System.Linq;

namespace Patterns.Factories
{

    public interface IPerson
    {

        int Id { get; set; }

        string Name { get; set; }

        //другие свойства
    }

    public interface IOrder
    {

        int Id { get; set; }

        int PersonId { get; set; }

        long Total { get; set; }
    }

    public interface IPersonService
    {

        IReadOnlyList<IPerson> GetPersonsByThisYear();

        //другие методы
    }
}
```

```
public interface IOrdersService
{
    IReadOnlyList<IOrder> GetOrdersByPerson(IPerson person);
}

public interface ICalculationService
{
    long CalculateOrders(IReadOnlyList<IOrder> orders);
}
}
```

И таких зависимостей может быть очень много, либо зависимости слишком усложнены дополнительной логикой, которая по факту конечному потребителю не нужна.


```

public sealed class ReportService
{
    private readonly IOrdersService _ordersService;

    private readonly ICalculationService _calculationService;

    private readonly IPersonService _personService;

    public ReportService(IOrdersService ordersService,
ICalculationService calculationService, IPersonService personService)
    {
        _ordersService = ordersService;

        _calculationService = calculationService;

        _personService = personService;
    }

    public void CreateReportForThisYear()
    {
        IReadOnlyList<IPerson> persons =
        _personService.GetPersonsByThisYear();

        List<IOrder> orders = new List<IOrder>();

        foreach (IPerson person in persons)
        {
            IReadOnlyList<IOrder> ordersByPersons =
            _ordersService.GetOrdersByPerson(person);

            orders.AddRange(ordersByPersons);
        }
    }
}

```

```
        _calculationService.CalculateOrders (orders);  
    }  
  
}
```

Смысл в том, что мы скрыли цепочку вызовов за одним методом. Об этой цепочки конечному потребителю знать не стоит, и она может и измениться. А вот контракт для потребителя не изменится нисколько. В этом большой плюс данного подхода.

Adapter

В самом начале материала рассматривался паттерн Decorator. Его часто путают с паттерном Adapter, но так как группа одних паттернов они действительно могут быть схожи, но есть отличия в нюансах. Если в декораторе мы добавили новый функционал, то тут мы берем совсем другой класс и адаптируем его под нужный нам тип, без добавления нового функционала. Это важная отличительная черта, тонкая красная линия различия этих двух паттернов.

```
using System;

using System.Collections.Concurrent;

using System.Collections.Generic;

using System.Linq;


namespace Patterns.Factories
{

    public interface IOrder
    {

        int Id { get; set; }

        long Total { get; set; }

    }


    public interface IOrderReportService
    {

        void CalculateReportByOrders(IReadOnlyList<IOrder> orders);

    }

}
```

Это самый простой пример. Некий сервис отчетов делает отчеты по коллекции объектов продаж. Модифицировать мы его не можем, он полностью где-то там далеко от нас и уже, как писалось выше замешан во многих местах логики. Но задача была поставлена так, чтобы новые типы продаж, к примеру оптовые, тоже подсчитывались данным сервисом.

```
public sealed class ComplexOrder
{
    public int Id { get; set; }

    public long TotalFirstPart { get; set; }

    public long TotalSecondPart { get; set; }
}
```

И самый быстрый способ — это сделать для данного объекта – адаптер, который будет отвечать неким «прокси» от функционала старого объекта до функционала нового.

```
public sealed class ComplexOrderAdapter : IOrder
{
    private readonly ComplexOrder _order;

    public ComplexOrderAdapter(ComplexOrder order)
    {
        _order = order;
    }

    public int Id
    {
        get
        {
            return _order.Id;
        }
        set
        {
            _order.Id = value;
        }
    }

    public long Total
    {
        get
        {
            return _order.TotalFirstPart + _order.TotalSecondPart;
        }
        set
        {
        }
    }
}
```

Мы не добавили новый функционал, именно скрыли его под старым. И соответственно использование данного класса теперь вполне пригодно для подсчета в сервисе отчетов.

Вывод

При написании фасадов, либо декораторов или адаптеров встречается, что названия данных паттернов добавляются к названию классов. Общего мнения на этот счет не сложилось. В некоторых компаниях считается дурным тоном давать название класса вместе с паттерном, а в других наоборот, считая, что код становится яснее.

Практическое задание

1. Придумайте небольшое приложение консольного типа, который берет различные Json структуры (предположительно из разных веб сервисов), олицетворяющие товар в магазинах. Структуры не похожи друг на друга, но вам нужно их учесть, сделать универсально. Структуры на ваше усмотрение.

Дополнительные материалы

1. Статья [«Шаблон проектирования»](#).

Используемые источники

1. Статья [«Шаблон проектирования»](#).
2. Статья [Design Patterns](#).