GeekBrains

ASP.NET MVC и углубление в изучение C#

Введение в паттерны. Порождающие

На этом уроке

1. Познакомимся с группами паттернов.
2. Поймём, для чего они используются.
3. Рассмотрим группу порождающих паттернов.
4. Разберём такие паттерны, как Abstract Factory, Factory Method и Builder.
5. Увидим разницу между Abstract Factory и Factory Method.

Оглавление

Введение в паттерны

Паттерны (шаблоны проектирования) — фундаментальная часть разработки программного обеспечения. С одной стороны, вы можете разрабатывать приложение и без знания паттернов и достичь цели, а с другой — знание паттернов позволит достичь цели проще и быстрее. Хотя и это не факт, так как некоторые паттерны требуют больше кода.

Поговорим об определении паттерна. В интернете есть много объяснений этому термину. В двух словах — это сборник общепринятых подходов реализации тех или иных участков кода и приложений. Впоследствии, если к вашему коду обратится другой разработчик, знающий паттерны, то разобраться в коде ему будет намного проще, так как он быстрее поймёт поведенческий мотив.

Но есть один неприятный момент, который ждёт практически каждого начинающего разработчика, вставшего на пути изучения паттернов — написание кода не ради цели, а ради паттернов. Это в корне неверный подход. Всегда повторяйте одно важное правило: «Паттерн — это не цель, а путь для достижения цели».

Впоследствии, получив больше опыта, применение паттернов войдёт у вас в автоматический режим, что вы даже не будете обращать на это внимание.

На предыдущих уроках вы уже сталкивались с некоторыми паттернами. Настало время углубиться в эту тему.

Группы паттернов

На собеседованиях часто спрашивают о знании паттернов. И первый вопрос, который вы можете услышать: «Какие паттерны есть?». Необязательно перечислять сами паттерны, стоит лишь назвать их группы, которых всего три.

Группа	Пример	Описание
Поведенческие	Chain of responsibility Strategy Visitor	Позволяю настраивать коммуникацию между разным кодом и объектами
Структурные	Adapter Façade Decorator	Предоставляют различные способы построения связей между кодом и объектами

Порождающие	Factory Method Abstract Factory Builder Singleton	Эффективные методы создания простых и сложных объектов, зачастую пряча оператор new от внешнего потребителя

Важно помнить, что в графе «пример» приведены самые популярные паттерны, а количество паттернов намного больше.

Зачастую на собеседованиях задаются совсем нелогичные вопросы о паттернах. Один из них — «а какие паттерны лучше?». Такие вопросы надо рассматривать как ловушку для новичков, чтобы проверить кандидата на общее понимание теории применения паттернов. Ответ на такой вопрос прост — всё зависит от ситуации и задачи. Лучших паттернов нет, зато есть хорошие решения при комбинации разных групп паттернов в решении задачи.

Например, в своём приложении вы используете много паттернов и даже один паттерн по многу раз. Допустим, Abstract Factory. Это действительно удобный паттерн, который позволяет потребителям не углубляться в специфичные знания создания интересующих объектов.

Поэтому нельзя сказать, что «приложение написано на архитектуре паттерна Builder». Воспользуйтесь другой формулировкой и скажите, что в архитектуре приложения вы применили паттерн Builder. Будьте внимательны на собеседованиях. Только через такие «нелогичные» вопросы, интервьюеры с первых слов понимают, разбирается кандидат в паттернах или нет.

Группа «Порождающие паттерны»

Одна из самых часто используемых групп паттернов при разработке. Именно они позволяют скрыть логику создания простых и сложных объектов для конечного потребителя. Это группа паттернов предоставляет контракт на создание объектов. И даже если логика создания объектов внутри реализации поменяется, то контракт продолжит работать.

То, когда их надо применять, сильно зависит от контекста выполняемой задачи, но, как правило, это объекты, которые создаются массово. Яркий пример — команды WPF.

Abstract Factory

Abstract factory (Абстрактная фабрика) — это паттерн, который позволяет эффективно создавать семейства каких-либо объектов. Он достаточно прост в реализации. Разберём его классическую реализацию и создадим сущности разных вымышленных отчётов. Для начала рассмотрим всех героев реализации.

1. Контракт различных накладных. Назовём его ISummaryItem.
2. Контракт специфичных накладных, унаследованных от ISummaryItem. Например, месячный отчёт и годовой — IMonthSummaryItem и IYearSummaryItem.
3. Контракт фабрики — ISummaryFactory с методами CreateMonthSummary и CreateYearSummary.

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

namespace Patterns.Factories
{
	public interface ISummaryItem
	{
	long Id { get; set; }

	long TotalMoney { get; }

	void Calculate();

	}

	public interface IMonthSummaryItem : ISummaryItem
	{
	//some specific logic
	}

	public interface IYearSummaryItem : ISummaryItem
	{
	//some specific logic
	}

	public interface ISummaryFactory
	{
	ISummaryItem CreateMonthSummary();

	ISummaryItem CreateYearSummary();
	}
}
```

На этом моменте нас не должна интересовать специфика отчётов. Важно, что это одно и то же семейство — ISummaryItem.

Предположим, что в компании, где будет применяться ваше программное обеспечение, есть n+1 отделов. И каждый отдел сформирует свои отчёты по месяцам и годам. У каждого отдела есть сложная логика расчёта этих накладных. Соответственно, метод расчёта отчётов будет сильно разниться от реализации к реализации. Но потребителям этих отчётов не важен отдел, главное для них — результат калькуляции.

```
using System;
```

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

namespace Patterns.Factories
{
    internal sealed class SummaryWorkerService
      {
      public void CalculateSummaryAndSaveTotalMoney(ISummaryItem summaryItem)
      {
            summaryItem.Calculate();

            long totalMoney = summaryItem.TotalMoney;
// etc...
      }
      }

}
```

Допустим, вам надо ввести отчётность в отдел продаж. И наш гипотетический отдел продаж считает отчётность по довольно простым формулам.

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

namespace Patterns.Factories
{
      internal sealed class SalesDepartmentMonthSummaryItem : IMonthSummaryItem
      {
      public long Id { get; set; }

      public long TotalMoney { get; private set; }

      public void Calculate()
      {
            DateTime now = DateTime.Now;

            int totalDays = DateTime.DaysInMonth(now.Year, now.Month);

            TotalMoney = totalDays * 50;
      }
      }

      internal sealed class SalesDepartmentYearSummaryItem : IYearSummaryItem
      {
      public long Id { get; set; }

      public long TotalMoney { get; private set; }
```

```
        public void Calculate()
        {
                TotalMoney = 12 * 500;
        }
        }

        public sealed class SalesDepartmentSummaryFactories : ISummaryFactory
        {
        public ISummaryItem CreateMonthSummary()
        {
                return new SalesDepartmentMonthSummaryItem();
        }

        public ISummaryItem CreateYearSummary()
        {
                return new SalesDepartmentYearSummaryItem();
        }
        }
}
```

Как можно понять из кода, мы спрятали специфическую логику и логику создания этих отчётов за фабрикой, открыв доступ только к ней.

Пример довольно простой, но смысл в том, что отделов может быть много, а калькуляция — различной. Это приводит к усложнению кода операторами new и открытию доступа к классам, которые по факту специфичны.

Factory Method

Один из самых простейших в понимании и реализации паттерн. Его сложность заключается в том, что люди часто путают его с абстрактной фабрикой. Но между ними есть чёткие различия, о которых речь пойдёт чуть дальше.

Фабричный метод — это такой метод, который создаёт объект контракта в объекте, где есть дополнительная и специфическая логика по отношению к конкретному объекту. Простыми словами — это метод, который создаёт объект в классе, где есть логика.

Разберём очередной пример. Допустим, есть некая общая логика для семейства объектов — чеки.

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

```
namespace Patterns.Factories
{

	public interface IBill
	{
	long Id { get; }
	}

	public abstract class BillManagement
	{
	protected abstract IBill CreateBill();

	public IBill RegisterNewBill()
	{
		IBill bill = CreateBill();

	     bool result = CommitBill(bill);

		if (result)
		{
		   Console.WriteLine($"Success to commit bill with id {bill.Id}");

		return bill;
		}

		throw new Exception("Failed to commit bill");

	}

	protected abstract bool CommitBill(IBill bill);
	}

}
```

Логика проста. Создаём чек и сохраняем его в каком-либо источнике данных, например, в базе данных. Но типов чеков может быть много, и для разных отделов компании возможна различная имплементация.

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

namespace Patterns.Factories
{
	internal sealed class SalesBill : IBill
	{
	private long _id;

	public long Id
```

```
        {
                get => _id * 100;
                set => _id = value;
        }
        }

        internal sealed class FinanceBill : IBill
        {
        private long _id;

        public long Id
        {
                get => _id * 1000;
                set => _id = value;
        }
        }

        public sealed class SalesBillManagement : BillManagement
        {
        protected override IBill CreateBill()
        {
                return new SalesBill();
        }

        protected override bool CommitBill(IBill bill)
        {
                //saves to database

                return true;
        }
        }

        public sealed class FinanceBillManagement : BillManagement
        {
        protected override IBill CreateBill()
        {
                return new FinanceBill();
        }

        protected override bool CommitBill(IBill bill)
        {
                //saves to database

                return true;
        }
        }
}
```

Разница между фабричным методом и абстрактной фабрикой

Это второй по популярности вопрос про паттерны на собеседовании. Ответ на него также неплохо показывает глубину знаний и понимание паттернов разработки. Разница между двумя этими паттернами довольно существенная.

1. Абстрактная фабрика — это комплексное решение создания семейства объектов для потребителя. То есть вы передаёте свою фабрику в использовании дальше по коду.
2. Фабричный метод — это больше локальное и гибкое создание семейства объектов внутри специфической логики. В редком случае это будет доступно для конечных потребителей.

Builder

Это третий по популярности использования паттерн в группе. Он ярко отличается от предыдущих тем, что позволяет упростить создание сложных объектов пошагово. Один из удобных примеров — реализация какого-либо сложного отчёта, который строится поэтапно, агрегируя какие-либо данные.

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

namespace Patterns.Factories
{

	public interface IPrintDevice
	{
	//some methods
	}

	public interface IReport
	{
	void Print(IPrintDevice printDevice);
	}

	internal sealed class BigReport : IReport
	{
	private readonly IList<string> _blocks = new List<string>();

	public void AddBlockInfo(string info)
	{
		_blocks.Add(info);
	}

	public void Print(IPrintDevice printDevice)
	{
```

```
            //some logic to print this report
        }
        }

        public sealed class ReportBuilder
        {
        private BigReport _report;

        public void AddHeader()
        {
              _report.AddBlockInfo("Adding header");
        }

        public void AddFooter()
        {
              _report.AddBlockInfo("Adding footer");
        }

        public void AddBody()
        {
              _report.AddBlockInfo("Adding body");
          }

        public void AddExInfo(string info)
        {
              _report.AddBlockInfo(info);
        }

        public void Reset()
        {
              _report = new BigReport();
        }

        public IReport BuildReport()
        {
              BigReport report = _report;

              Reset();

              return report;
        }
        }
}
```

Мы видим довольно гибкое создание отчёта. Потребитель может добавить заголовок или свою информацию, так как такая возможность есть.

Практическое задание

1. Используя знания о паттернах, выберите приглянувшийся вам и создайте ICommand для WPF.

Дополнительные материалы

1. Статья «Шаблон проектирования».

Используемые источники

1. Статья «Шаблон проектирования».

2. Статья Design Patterns.