GeekBrains

ASP.NET MVC и углубление в изучение C#

Введение в паттерны. Поведенческие

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 Visitor, Strategy и Chains of Responsibility.
5. Увидим разницу между данными паттернами.

Оглавление

Введение в паттерны 3

Заключительная часть про паттерны пройдет по одной из самых сложных групп – поведенческие. Наравне с другими группами паттернов данная группа так же часто используема и почитаема среди разработчиков. Даже при работе с внутренними библиотеками .net можно встретиться с их реализацией на примере. К примеру, события event это ничто иное, как один из видов реализации паттерна Observer. Middleware в asp.net core является реализацией паттерна Chains of Responsibility. Если присмотреться с высоты птичьего полета, именно эта группа паттернов больше всего сосредоточена в бизнес-логике разрабатываемого программного продукта, так как идеально помогает настраивать коммуникацию среди ваших логических частей кода. Отсюда и название – поведенческие паттерны.

Группа Поведенческие паттерны

Как и обычно, некоторые поведенческие паттерны можно спутать с другими паттернами из относительно схожей группы. К примеру, паттерн Strategy является великолепным тому примером, за исключением того, что данный паттерн заточен под то, чтобы динамически менять поведение бизнес-логики исходя из контекста. Некоторые паттерны, которые работают из коробки, в .net могут даже несколько усугубить ситуацию. Пример: event подписки (реализация Observer) могут привести к утечке памяти, если совершать подписку на событие и в дальнейшем не отписываться от него.

Будьте всегда осторожны при написании кода! Так как некоторые ошибки могут стоить дорого – как минимум потери времени, как тестировщика, так и разработчика. А именно человеко-часами измеряется стоимость разработки продукта.

Паттерн Strategy

Как и говорилось выше, паттерн Strategy является удобным паттерном, который в зависимости от потребности потребителя, может менять поведение кода в целом. И стоит помнить, что это он меняется в динамическом режиме – на лету.

Представим гипотетический пример. Стоит задача на создание модуля по работе с каким-либо сканером. Сканер может сканировать документы, фотографии и все, что допустимо. Но на выходе нужно формировать различные форматы – картинки, распознанные документы в формате docx, pdf документы. И данный список будет только увеличиваться по мере использования данного сканера. А также сканер может понадобиться для еще более специфичной задачи. Но нас, как разработчиков не

должно это смущать. Мы должны предугадать этот момент и сделать поддержку новых форматов с минимальным временем внесения новых форматов.

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


namespace Patterns.Factories
{
   public interface IScannerDevice
  {
      Stream Scan();
  }
}
```

Допустим это интерфейс устройства сканера, который на выходе выдает только некий Stream с потоком байтов. Как же можно применить паттерн Strategy для поддержки n+1 форматов? Достаточно просто и легко.

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

namespace Patterns.Factories
{
   public interface IScannerDevice
  {
    Stream Scan();
  }
   public sealed class ScannerContext
  {
      private readonly IScannerDevice _device;
      private IScanOutputStrategy _currentStrategy;

      public ScannerContext(IScannerDevice device)
      {
      _device = device;
      }

      public void SetupOutputScanStrategy(IScanOutputStrategy strategy)
      {
      _currentStrategy = strategy;
      }

      public void Execute(string outputFileName = "")
```

```
      {
      if (_device is null)
      {
      throw new ArgumentNullException("Device can not be null");
      }
      if (_currentStrategy is null)
      {
      throw new ArgumentNullException("Current scan strategy can not be
null");
      }
       if (string.IsNullOrWhiteSpace(outputFileName))
      {
      outputFileName = Guid.NewGuid().ToString();
      }
      _currentStrategy.ScanAndSave(_device, outputFileName);
      }
  }


  public interface IScanOutputStrategy
  {
      void ScanAndSave(IScannerDevice scannerDevice, string
outputFileName);
  }


  public sealed class PdfScanOutputStrategy : IScanOutputStrategy
  {
      public void ScanAndSave(IScannerDevice scannerDevice, string
outputFileName)
      {
```

```
      //do pdf output
      }
  }

  public sealed class ImageScanOutputStrategy : IScanOutputStrategy
  {
      public void ScanAndSave(IScannerDevice scannerDevice, string
outputFileName)
      {
      //do image outptut
      }
  }
}
```

По коду можно увидеть, что создаётся некий контекст устройства, который может принимать на вход реализацию контракта IScanOutputStrategy, сам контракт устройства сканирования IScannerDevice и производить процедуру сканирования и сохранения. В зависимости от потребности, стратегию сканирования можно подменять другими реализациями на лету. Это и добавляет высокой гибкости данному коду и переживать, о том, что нужно будет поддержать новые форматы в будущем уже не так сильно.

Паттерн Chain of Responsibility

Один из важнейших паттернов данной группы. Именно он позволяет создать некий pipe (пайп), который постепенно и поэтапно будет обрабатывать потоки данных. Он так же вполне может быть динамически настроен, и реализация его в .net можно встретить в различных библиотеках. Опять же, обратимся к гипотетическому примеру. Есть некое устройство, как в предыдущем примере, которое выдает поток данных в виде неких структур данных. Эти данные, которые нужно обрабатывать либо по-разному, либо следить, чтобы цепочка вызовов оборвалась, по какой-либо причине.

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

namespace Patterns.Factories
{
	public interface IMonitorData
	{
	int Cpu { get; }
	int Voltage { get; }
	bool TurnedOn { get; }
	}

	public interface IMonitoringSystemDevice
	{
	IEnumerator<IMonitorData> GetMonitorData();
	}
}
```

Данные простые – метрики процессора, вольтаж и маркер, показывающий, что устройство включено. И как уже было сказано выше, нужно эти данные обрабатывать по-разному. Для этого стоит использовать, как раз паттерн «Цепочка Обязанностей». Создадим контракты.

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

namespace Patterns.Factories
{
	public interface IMonitorData
	{
	int Cpu { get; }
	int Voltage { get; }
	bool TurnedOn { get; }
	}

	public interface IMonitoringSystemDevice
	{
	IEnumerator<IMonitorData> GetMonitorData();
	}

	public interface IMonitorPipelineItem
	{
	void SetNextItem(IMonitorPipelineItem pipelineItem);

	void ProcessData(IMonitorData data);
	}

	public abstract class MonitorPipelineItem : IMonitorPipelineItem
```

```
	{
	private IMonitorPipelineItem _next;

	public void SetNextItem(IMonitorPipelineItem pipelineItem)
	{
		_next = pipelineItem;
	}

	public void ProcessData(IMonitorData data)
	{
		if (ReviewData(data) && _next != null)
		{
		_next.ProcessData(data);
		}
	}

	protected abstract bool ReviewData(IMonitorData data);
	}
}
```

Стоит обратить внимание, что, что контракт IMonitorPipeline принимает следующего в очереди по вызову участника обработки входящих данных.

И конечная реализация данного паттерна является довольно простой. Приведен полный пример, чтобы проще было разобраться с логической цепочкой.

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

namespace Patterns.Factories
{
	public interface IMonitorData
	{
	int Cpu { get; }
	int Voltage { get; }
	bool TurnedOn { get; }
	}

	public interface IMonitoringSystemDevice
	{
	IEnumerator<IMonitorData> GetEnumerator();
	}


	public interface IMonitorPipelineItem
	{
	void SetNextItem(IMonitorPipelineItem pipelineItem);

	void ProcessData(IMonitorData data);
	}
```

```
public abstract class MonitorPipelineItem : IMonitorPipelineItem
{
private IMonitorPipelineItem _next;

public void SetNextItem(IMonitorPipelineItem pipelineItem)
{
      _next = pipelineItem;
}

  public void ProcessData(IMonitorData data)
{
      if (ReviewData(data) && _next != null)
      {
      _next.ProcessData(data);
      }
}

protected abstract bool ReviewData(IMonitorData data);
}

public sealed class CpuMonitorPipelineItem : MonitorPipelineItem
{
protected override bool ReviewData(IMonitorData data)
{
      if (data is null)
      {
      return false;
      }
```

```
        if (data.Cpu < 2)
        {
        return false;
        }

        //do some work

        return true;
    }
    }

    public sealed class VoltageMonitorPipeline : MonitorPipelineItem
    {
    protected override bool ReviewData(IMonitorData data)
    {
        if (data is null)
        {
        return false;
        }

        if (data.Voltage == 0)
        {
        return false;
        }

        //do some work
```

```
            return true;
      }
      }
      public sealed class MonitorDeviceContext
      {
      private readonly IMonitoringSystemDevice _monitoringSystemDevice;

      public                  MonitorDeviceContext(IMonitoringSystemDevice
monitoringSystemDevice)
      {
            _monitoringSystemDevice = monitoringSystemDevice;
      }

      public void RunMonitorProcess()
      {
            IMonitorPipelineItem pipelineItem = CreatePipeline();

            foreach (IMonitorData data in _monitoringSystemDevice)
            {
                pipelineItem.ProcessData(data);
            }
      }

      private IMonitorPipelineItem CreatePipeline()
      {
            IMonitorPipelineItem     cpuMonitorPipelineItem     =     new
CpuMonitorPipelineItem();
            IMonitorPipelineItem    voltageMonitorPipelineItem    =    new
VoltageMonitorPipeline();
```

```
cpuMonitorPipelineItem.SetNextItem(voltageMonitorPipelineItem);

            return cpuMonitorPipelineItem;
      }
      }
}
```

Создаются два элемента цепочки, которые обрабатывают данные поэтапно. Первый элемент обрабатывает данные с процессора, второй – с вольтажа. Каждый из них можно прервать по какому-либо условию и также можно продолжить дополнять вызов этой цепочки. Это абсолютно классическое представление данного паттерна. Но есть и упрощенный вариант, когда при использовании контейнера, к примеру Autofac, в конструктор можно передать список обработчиков с полями, которые идентифицируют его позицию в очереди вызовов и обычным перебором могут вызывать эти обработчики на полученных данных.

Это действительно хороший паттерн, логически грамотно разделяющий ваш код на маленькие части, которые будут заниматься специфической логикой, не размазывая код на большие классы с тоннами методов.

Паттерн Visitor

Если спросить, какие самые известные паттерны есть, то паттерн Visitor, наверно будет вторым в списке по известности. Это сложный паттерн и стоит его применять в крайней необходимости. Его достаточно часто применяют для обхода каких-либо графов. Разберем простейший пример. Допустим, как в предыдущем примере есть некие устройства, которое выдают информацию о процессоре и загрузке памяти.

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

namespace Patterns.Factories
{
    public interface ICpuData
   {
      int Percent { get; }
      int Threads { get; }
      bool Error { get; }
   }

   public interface IRAMMemory
   {
      int FreeMem { get; }
      int TotalMem { get; }
      bool Error { get; }
   }
   public interface IDeviceInfo
   {
      ICpuData Cpu { get; }

      IRAMMemory Memory { get; }
   }
}
```

Конкретно нас интересует его контракт-агрегатор IDeviceInfo, который уже содержит в себе и информацию о процессоре и информацию о памяти. Задача заключается в том, что нужно мониторить и память, и процессор, но с сильно различной логикой. Для этого стоит добавить контракт визитора, и несколько реализаций устройств с новым методом принятия этого визитора.

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

namespace Patterns.Factories
{

   public interface ICpuData
   {
      int Percent { get; }
      int Threads { get; }
      bool Error { get; }
   }

   public interface IRAMMemory
   {
      int FreeMem { get; }
      int TotalMem { get; }
      bool Error { get; }
   }

   public interface IDeviceInfo
   {
      ICpuData Cpu { get; }

      IRAMMemory Memory { get; }
```

```
    void Accept(IMonitorVisitor visitor);
}

public sealed class ComputerDeviceInfo : IDeviceInfo
{
    public ICpuData Cpu { get; set; }

    public IRAMMemory Memory { get; set; }
    public void Accept(IMonitorVisitor visitor)
    {
    if (visitor is null)
    {
          return;
    }

    visitor.VisitComputer(this);
     }
}

public sealed class ControllerDeviceInfo : IDeviceInfo
{
    public ICpuData Cpu { get; set; }

    public IRAMMemory Memory
    {
    get { throw new NotImplementedException(); }
    }
```

```
    public void Accept(IMonitorVisitor visitor)
    {
    if (visitor is null)
    {
          return;
    }

    visitor.VisitController(this);
    }
}

public interface IMonitorVisitor
{
    void VisitComputer(IDeviceInfo info);
    void VisitController(IDeviceInfo info);
}

public sealed class DeviceVisitor : IMonitorVisitor
{
    public void VisitComputer(IDeviceInfo info)
    {
    //do some work
    }

    public void VisitController(IDeviceInfo info)
    {
    //do some work
```

```
        }
    }
}
```

Можно заметить, что паттерн немного запутанный. По факту, само устройство, а точнее конкретная реализация устройства сама выбирает, какой метод у визитора вызывать. Это не всегда удобно, но об этом методе стоит помнить, когда другой обход сложен, либо невозможен.

Практическое задание

1. Сделайте эмулятор устройства сканера. Он сканирует (берет данные из какого либо файла), производит фейковые данные о загрузке процессора и памяти. Код должен быть прост, и дальнейшую работу стоит вести только с контрактами данного устройства. Разработать небольшую библиотеку, которая принимает от этого эмулятора байты, сохраняет в различные форматы и мониторит его состояние, записывая в какой-либо лог.

Дополнительные материалы

1. Статья «Шаблон проектирования».

Используемые источники

1. Статья «Шаблон проектирования».
2. Статья Design Patterns.