Занятие 7. Структурные паттерны

Теоретическая часть

Структурные паттерны проектирования - это паттерны, которые помогают структурировать иерархию классов и объектов.

Наиболее часто используемые структурные паттерны:

- Adapter - преобразует интерфейс одного класса в интерфейс другого, чтобы они могли работать вместе.
- Bridge - разделяет абстракцию и реализацию так, чтобы они могли изменяться независимо.
- Composite - создает древовидную структуру из объектов.
- Decorator - добавляет новые функциональные возможности к объекту без изменения его исходного кода.
- Facade - предоставляет простой интерфейс к сложной системе.
- Flyweight - экономит память, разделяя данные между объектами.

Практическая часть

Для рассморения структурных паттернов воспользуемся примером из занятия 2 с применением принципов SOLID.

Подготовка проекта

Создадим проект следующими командами:

```
dotnet new solution -n 07-structural-patterns
dotnet sln migrate
rm 07-structural-patterns.sln
dotnet new console -n AppWithStructuralPatterns -o ./AppWithStructuralPatterns
dotnet sln add ./AppWithStructuralPatterns/AppWithStructuralPatterns.csproj
```

После создания проекта скопируем код проекта `AppWithSolid` из занятия 2, а именно:

- `Models/Report.cs`
- `Services/IReportSaver.cs`
- `Services/Implementations/TextReportSaver.cs`
- `Services/IReportSender.cs`
- `Services/Implementations/EmailReportSender.cs`
- `Services/ReportService.cs`

И изменим корневое пространство имен на `AppWithStructuralPatterns`.

Анализ имеющегося кода

Для начала проанализируем имеющийся код. Если посмотреть внимательно, то можно заметить, что один из структурных паттернов уже присутствует в коде - это Facade. Его реализацию можно увидеть в классе `ReportService`.

Чтобы понять, как работает данный паттерн, давайте рассмотрим его подробнее. Суть патерна Facade состоит в том, чтобы предоставить простой интерфейс к сложной системе. Посмотрим на код класса `ReportService`:

```
public sealed class ReportService(IReportSaver reportSaver, IReportSender
reportSender)
{
    public void ProcessReport(Report report)
    {
        reportSaver.SaveReport(report);
        reportSender.SendReport(report);
    }
}
```

Как мы видим, класс `ReportService` использует два сервиса: `IReportSaver` и `IReportSender`. При этом данный класс скрывает их использование от вызывающего кода - для вызывающего кода использование этих сервисов представлено в виде одной операции "Обработать отчет". Если мы захотим добавить дополнительный этап в процесс обработки отчета, то для вызывающего кода при этом ничего не изменится - он будет продолжать использовать только одну операцию "Обработать отчет".

В этом и состоит суть паттерна Facade - скрыть сложную систему за простым интерфейсом.

Реализация паттерна Adapter

Для реализации паттерна Adapter представим, что нам необходимо поработать со сторонней библиотекой, умеющей сохранять отчеты в формате XML. В качестве этой библиотеки будем использовать новый проект `ThirdPartyLibrary`.

Создание библиотеки

Создадим проект библиотеки следующими командами:

```
dotnet new classlib -n ThirdPartyLibrary -o ./ThirdPartyLibrary
dotnet sln add ./ThirdPartyLibrary/ThirdPartyLibrary.csproj
```

А теперь добавим ссылку на данную библиотеку в проект `AppWithStructuralPatterns`:

```
dotnet add ./AppWithStructuralPatterns/AppWithStructuralPatterns.csproj reference
./ThirdPartyLibrary/ThirdPartyLibrary.csproj
```

Теперь добавим в корень проекта `ThirdPartyLibrary` класс `Report` для представления отчетов в нашей библиотеке:

```
namespace ThirdPartyLibrary;

public sealed record Report(
    string Title,
    DateTime Date,
    IReadOnlyCollection<string> Rows
);
```

Как видим, класс `Report` из библиотеки существенно отличается от класса `Report` нашего приложения.

Далее добавим в корень библиотеки класс `XmlReportSaver` для сохранения отчетов в формате XML:

```
using System.Xml;

namespace ThirdPartyLibrary;

public static class XmlReportSaver
{
    public static void SaveReport(Report report, string fileName)
    {
        if (string.IsNullOrEmpty(fileName))
        {
            throw new ArgumentNullException(nameof(fileName));
        }

        var xml = new XmlDocument();

        var reportNode = xml.CreateElement("report");

        var titleNode = xml.CreateElement("title");
        titleNode.InnerText = report.Title;
        reportNode.AppendChild(titleNode);

        var dateNode = xml.CreateElement("date");
        dateNode.InnerText = report.Date.ToString("yyyy-MM-dd");
        reportNode.AppendChild(dateNode);

        foreach (var row in report.Rows)
        {
            var rowNode = xml.CreateElement("row");
            var valueNode = xml.CreateElement("value");
            valueNode.InnerText = row;
            rowNode.AppendChild(valueNode);
            reportNode.AppendChild(rowNode);
        }

        xml.AppendChild(reportNode);
```

```
        xml.Save(fileName);
    }
}
```

В коде выше мы видим код формирования XML-документа и его сохранения в файл. В данном материале мы не будем подробно останавливаться на рассмотрении данного кода, так как он не является темой данного занятия. Подробнее с API .NET для работы с XML можно ознакомиться по ссылке.

Реализация адаптера

Теперь, когда у нас есть библиотека, умеющая сохранять отчеты в формате XML, нам необходимо внедрить ее использование в наше основное приложение.

Для этого добавим в каталог `Services/Implementations` проекта `AppWithStructuralPatterns` класс `XmlReportSaverAdapter` для адаптации интерфейса `IReportSaver` к интерфейсу `XmlReportSaver`:

```
using AppWithStructuralPatterns.Services;
using AppWithStructuralPatterns.Models;
using ThirdPartyLibrary;

using Report = AppWithStructuralPatterns.Models.Report;
using ForeignReport = ThirdPartyLibrary.Report;

internal sealed class XmlReportSaverAdapter(string fileName) : IReportSaver
{
    public void SaveReport(Report report)
    {
        var rows = new[] {
            $"Количество проданных автомобилей: {report.CarsSold}",
            $"Количество проданных мотоциклов: {report.MotorcyclesSold}",
        };

        var foreignReport = new ForeignReport(
            Title: report.Title,
            Date: report.Date.ToDateTime(report.Time),
            Rows: rows
        );

        XmlReportSaver.SaveReport(foreignReport, fileName);
    }
}
```

Как видим, наш новый класс адаптирует классы из сторонней библиотеки к интерфейсам, используемым в нашем приложении - поэтому данный паттерн и называется Adapter.

Из нового в данном коде можно увидеть пример использования псевдонимов для типов - данная функциональность языка позволяет нам именовать уже имеющиеся типы другим образом для

использования в текущем файле. Чаще всего, это необходимо при наличии нескольких классов с одинаковыми называниями, как, например, в нашем случае.

Реализация паттерна Composite

Теперь представим, что при обработке отчетов мы хотим отправлять их не только по электронной почте, но и, допустим, в Telegram.

Если решать задачу в лоб, то мы могли бы либо добавить новый сервис для отправки в Telegram и использовать его в `ReportService`, либо добавить возможность использования массива Sender'ов в `ReportService`. Но первый вариант плох тем, что при добавлении нового способа отправки нам нужно будет снова менять код `ReportService`. Второй подход лучше, так как позволяет в будущем легко расширять список способов доставки отчета, но при этом он все равно затрагивает код `ReportService`, а подобного в условиях совместной работы над проектами лучше избегать.

В данном случае на выручку приходит паттерн Composite.

Паттерн используется для представления древовидной структуры объектов. В нашем случае корнем дерева будет реализация `IReportSender`, умеющая перенаправлять вызовы в дочерние Sender'ы.

Для начала создадим в каталоге `Services/Implementations` нашего проекта класс `CompositeReportSender`:

```
using AppWithStructuralPatterns.Models;
using AppWithStructuralPatterns.Services;

namespace AppWithStructuralPatterns.Services.Implementations;

internal sealed class CompositeReportSender(params
IReadOnlyCollection<IReportSender> senders) : IReportSender
{
    public void SendReport(Report report)
    {
        foreach (var sender in senders)
        {
            sender.SendReport(report);
        }
    }
}
```

Как видим, класс `CompositeReportSender` является простой оберткой над массивом Sender'ов, которая позволяет использовать их как единый объект.

Теперь добавим реализацию интерфейса `IReportSender` для отправки отчета в Telegram. Так как интеграция с Telegram не является темой данного занятия, то мы просто будем выводим в консоль соответствующее сообщение.

Добавим класс `TelegramReportSender` в каталог `Services/Implementations` проекта `AppWithStructuralPatterns`:

```
using AppWithStructuralPatterns.Services;
using AppWithStructuralPatterns.Models;

namespace AppWithStructuralPatterns.Services.Implementations;

internal sealed class TelegramReportSender(string chatId) : IReportSender
{
    public void SendReport(Report report)
    {
        Console.WriteLine($"Отправка отчета в Telegram: {chatId}");
    }
}
```

Реализация паттерна Decorator

Теперь представим, что мы хотим добавить возможность логирования вызовов методов `SendReport` для всех Sender'ов.

Можно было бы решить задачу простым способом, добавив соответствующий код в каждый из Sender'ов. Но такой подход плох тем, что, во-первых, нам нужно менять уже существующий работающий код, а, во-вторых, мы усложняем добавление новых Sender'ов в будущем.

Поэтому практичнее будет использовать паттерн Decorator. Паттерн Decorator похож на рассмотренный ранее паттерн Composite, но отличается тем, что оборачивает только один объект, а не массив объектов.

Рассмотрим на практике реализацию данного паттерна. Добавим в каталог `Services/Implementations` нашего проекта класс `LoggingReportSenderDecorator`:

```
using AppWithStructuralPatterns.Services;
using AppWithStructuralPatterns.Models;

namespace AppWithStructuralPatterns.Services.Implementations;

internal sealed class LoggingReportSenderDecorator<TSender>(TSender sender) :
IReportSender
    where TSender : IReportSender
{
    public void SendReport(Report report)
    {
        Console.WriteLine($"{typeof(TSender).Name}: Начало отправки отчета:
{report.Title}");
        sender.SendReport(report);
        Console.WriteLine($"{typeof(TSender).Name}: Конец отправки отчета:
{report.Title}");
    }
}

internal static class LoggingReportSenderDecoratorExtensions
{
```

```
    public static IReportSender AddLogging<TSender>(this TSender sender)
        where TSender : IReportSender
    {
        return new LoggingReportSenderDecorator<TSender>(sender);
    }
}
```

Как видим, класс `LoggingReportSenderDecorator` оборачивает Sender и добавляет логирование вызовов метода `SendReport` в консоль.

Из нового здесь можно увидеть использование так называемых generic constraints - это выражения, ограничивающие типы, которые могут быть использованы в качестве аргументов для generic-параметров.

Также после объявления класса декоратора мы также видим добавление метода-расширения `AddLogging` для всех реализаций интерфейса `IReportSender`. Данный метод при вызове будет возвращать новую реализацию интерфейса `IReportSender` с подключенным логированием.

Использование получившегося кода

Теперь, когда у нас есть все необходимые компоненты, мы можем использовать их в `Program.cs`:

```
using AppWithStructuralPatterns.Models;
using AppWithStructuralPatterns.Services;
using AppWithStructuralPatterns.Services.Implementations;

var xmlReportSaver = new XmlReportSaverAdapter("report.xml");
var emailReportSender =  new
EmailReportSender("example@example.com").AddLogging();
var telegramReportSender = new TelegramReportSender("1234567890").AddLogging();
var compositeReportSender = new CompositeReportSender(emailReportSender,
telegramReportSender);

var reportService = new ReportService(xmlReportSaver, compositeReportSender);

var report = new Report(
    Title: "Отчет",
    Date: DateOnly.FromDateTime(DateTime.Now),
    Time: TimeOnly.FromDateTime(DateTime.Now),
    CarsSold: 100,
    MotorcyclesSold: 50
);

reportService.ProcessReport(report);
```

Запустим приложение:

```
dotnet run --project ./AppWithStructuralPatterns/AppWithStructuralPatterns.csproj
```

Если все сделано верно, то в текущем каталоге должен появиться файл `report.xml`, а в консоли должны появиться добавленные нами сообщения о начале и конце отправки отчета.

Как видим, благодаря структурным паттернам, мы смогли легко расширить функционал нашего приложения без изменения кода уже добавленных ранее компонентов.