Занятие 3. Инверсия управления

Теоретическая часть

Инверсия управления (Inversion of Control, IoC) - это принцип проектирования, который позволяет уменьшить связанность между классами и повысить гибкость и модульность кода.

Зачем это нужно:

- Уменьшение связанности между классами позволяет вносить изменения более точечно, не затрагивая другие части кода. Это облегчает поддержку кода, его тестирование, а также взаимодействие с другими разработчиками, так как чем меньше кода изменяется, тем меньше вероятность получить merge-конфликты.
- Увеличение модульности кода позволяет повысить переиспользование кода, а также тестировать одни компоненты изолированно от других, что в свою очередь позволяет повысить надежность кода и сэкономить время и деньги.

На практике можно встретить два подхода к инверсии управления:

- Service Locator - это паттерн, который позволяет получить доступ к сервисам через специальный класс, который хранит ссылки на все сервисы.
- Dependency Injection - это паттерн, который позволяет передать зависимость в класс через конструктор, свойство или метод.

В данном занятии мы рассмотрим оба подхода.

Практическая часть

Service Locator

Для рассмотрения на практике воспользуемся примером с соблюдением принципов SOLID из занятия 2, но добавим использование Service Locator.

Создание проекта

Создадим проект следующими командами:

```
dotnet new solution -n 03-ioc
dotnet sln migrate
rm 03-ioc.sln
dotnet new console -n AppWithServiceLocator -o ./AppWithServiceLocator
dotnet sln add ./AppWithServiceLocator/AppWithServiceLocator.csproj
```

Создание класса `ServiceLocator`

Создадим класс `ServiceLocator` в корне проекта:

```
using System.Collections.Generic;

namespace AppWithServiceLocator;

internal static class ServiceLocator
{
    private static readonly Dictionary<Type, object> _services = [];

    public static void AddService<T>(T service)
        where T : notnull
    {
        _services[typeof(T)] = service;
    }

    public static T? GetService<T>()
    {
        return _services.TryGetValue(typeof(T), out var service) ? (T)service :
default;
    }
}
```

Как мы видим, класс `ServiceLocator` является статическим классом, который позволяет:

- Добавлять сервисы в контейнер
- Получать сервисы из контейнера по типу

Добавление остального кода

Скопируем остальной код из занятия 2, а именно:

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

И изменим корневое пространство имен на `AppWithServiceLocator`.

Создание фасада для работы с отчетами

Мы не будем копировать код `ReportService`, а напишем новый с использованием Service Locator.

Добавим класс `ReportService` в папке `Services` проекта:

```
using AppWithServiceLocator.Models;

namespace AppWithServiceLocator.Services;

public sealed class ReportService
{
```

```
    public void ProcessReport(Report report)
    {
        var reportSaver = ServiceLocator.GetService<IReportSaver>();
        var reportSender = ServiceLocator.GetService<IReportSender>();

        reportSaver?.SaveReport(report);
        reportSender?.SendReport(report);
    }
}
```

Как мы видим, у нас отпала необходимость передавать сервисы в конструктор класса `ReportService`. При этом мы всё также можем менять форматы сохранения и отправки отчетов, не затрагивая код `ReportService`.

Но сразу же можно обратить внимание и на минусы данного подхода:

- Наш класс жестко завязан на класс `ServiceLocator`. Если мы захотим использовать другой контейнер зависимостей, то нам придется изменить код класса `ReportService`.
- Не заглядывая в код класса `ReportService`, мы не можем понять, какие зависимости он использует - это затрудняет понимание кода и тестирование.

Код `Program.cs`

Теперь объединим все наши компоненты в `Program.cs`:

```
using AppWithServiceLocator;
using AppWithServiceLocator.Models;
using AppWithServiceLocator.Services;
using AppWithServiceLocator.Services.Implementations;

ServiceLocator.AddService<IReportSaver>(new TextReportSaver("report.txt"));
ServiceLocator.AddService<IReportSender>(new
EmailReportSender("example@example.com"));

var reportService = new ReportService();
var report = new Report(
    Title: "Отчет",
    Date: DateTime.Now.ToString("dd.MM.yyyy"),
    Time: DateTime.Now.ToString("HH:mm:ss"),
    CarsSold: 100,
    MotorcyclesSold: 50
);
reportService.ProcessReport(report);
```

Из нового кода мы видим, что мы больше не добавляем сервисы в конструктор класса `ReportService`, а добавляем их в `ServiceLocator`.

При этом, если какому-то другому классу понадобятся сервисы, которые мы уже зарегистрировали, то нам также не нужно будет дополнительно передавать их в конструктор.

Запуск приложения

Запустим приложение:

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

Как мы видим, результат работы приложения не изменился.

Dependency Injection

Теперь рассмотрим использование Dependency Injection. В занятии 2 мы уже написали код, готовый к использованию совместно с DI, но там мы вручную создавали экземпляры сервисов. Теперь рассмотрим более продвинутый вариант с автоматическим созданием экземпляров сервисов. Для этого мы воспользуемся реализацией DI-контейнера от Microsoft. Он распространяется в виде двух пакетов:

- `Microsoft.Extensions.DependencyInjection` - основной пакет, который содержит основные интерфейсы и классы для работы с DI. Обычно данный пакет применяется в основном сборке приложения, которая создает и управляет DI-контейнером.
- `Microsoft.Extensions.DependencyInjection.Abstractions` - пакет, который содержит абстракции для работы с DI. Обычно данный пакет применяется в сборках приложения, которым нужно взаимодействовать с DI-контейнером, но не нужно создавать и управлять им.

Создание проекта

Решение у нас уже создано - поэтому создадим только проект:

```
dotnet new console -n AppWithDependencyInjection -o ./AppWithDependencyInjection
dotnet sln add ./AppWithDependencyInjection/AppWithDependencyInjection.csproj
```

Добавление пакетов

Теперь добавим в проект пакет `Microsoft.Extensions.DependencyInjection`. Пакет с абстракциями мы не добавляем, так как он нам не понадобится.

```
dotnet add ./AppWithDependencyInjection/AppWithDependencyInjection.csproj package
Microsoft.Extensions.DependencyInjection
```

Добавление кода компонентов

Скопируем код компонентов 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 `AppWithDependencyInjection`.

Объединение кода

Объединим все компоненты в `Program.cs`:

```
using Microsoft.Extensions.DependencyInjection;
using AppWithDependencyInjection.Models;
using AppWithDependencyInjection.Services;
using AppWithDependencyInjection.Services.Implementations;

var services = new ServiceCollection();

services.AddSingleton<IReportSaver>(_ => new TextReportSaver("report.txt"));
services.AddSingleton<IReportSender>(_ => new
EmailReportSender("example@example.com"));
services.AddSingleton<ReportService>();

var provider = services.BuildServiceProvider();

var reportService = provider.GetRequiredService<ReportService>();

var report = new Report(
    Title: "Отчет",
    Date: DateTime.Now.ToString("dd.MM.yyyy"),
    Time: DateTime.Now.ToString("HH:mm:ss"),
    CarsSold: 100,
    MotorcyclesSold: 50
);

reportService.ProcessReport(report);
```

Как мы видим, код стал более громоздким. Рассмотрим его подробнее.

В первой секции мы инициализируем DI-контейнер. Инициализация начинается с создания коллекции сервисов при помощи строки:

```
var services = new ServiceCollection();
```

В дальнейшем все сервисы регистрируются в данной коллекции. Для этого могут быть использованы различные методы, смысл которых мы рассмотрим в дальнейшем, когда коснемся разработки веб-приложений. Сейчас мы воспользуемся методом `AddSingleton` - при его использовании у нас регистрируется всего один экземпляр сервиса, который будет существовать на протяжении всего времени существования DI-контейнера.

Далее мы регистрируем наши сервисы при помощи коды:

```
services.AddSingleton<IReportSaver>(_ => new TextReportSaver("report.txt"));
services.AddSingleton<IReportSender>(_ => new
EmailReportSender("example@example.com"));
services.AddSingleton<ReportService>();
```

Как можно увидеть, мы регистрируем абсолютно все сервисы, которые в дальнейшем нам могут понадобиться. Если мы не зарегистрируем какой-то сервис, то DI-контейнер не сможет построить либо данный сервис, либо сервис, который от него зависит.

После регистрации сервисов следует этап построения DI-контейнера при помощи кода:

```
var provider = services.BuildServiceProvider();
```

Объект в переменной `provider` далее используется для запроса экземпляров сервисов.

В остальном код остался таким же, как и в случае с Service Locator.

Запуск приложения

Запустим приложение:

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

Как мы видим, результат работы приложения не изменился.

Сравнение подходов

В данном занятии мы рассмотрели два подхода к инверсии управления: Service Locator и Dependency Injection. Давайте сравним их преимущества и недостатки.

Service Locator:

- Преимущества:
  - Простота настройки
  - Возможность использования в любой части кода
- Недостатки:
  - Жесткая зависимость от класса `ServiceLocator`
  - Нет возможности понять, какие зависимости использует класс

Dependency Injection:

- Преимущества:
  - Возможность понять, какие зависимости использует класс

    - Нет зависимости от DI-контейнера - код более переносимый
    - В любом момент при необходимости можно перейти к Service Locator
- Недостатки:
    - Более сложная настройка

Все плюсы и минусы здесь очевидны, но внимательный читатель может обратить внимание на последний плюс DI, в котором сказано, что в любом момент при необходимости можно перейти к Service Locator. На практике это работает так, что DI-контейнер от Microsoft позволяет внедрять специальные зависимости: `IServiceProvider` и `IServiceScopeFactory`. Данные зависимости позволяют динамически запрашивать необходимые нам сервисы из DI-контейнера без предварительного объявления их в конструкторе.

В качестве итога стоит отметить, что ни один из паттернов не является идеальным и для простых приложений практичнее применять Service Locator из-за его простоты и наглядности, когда как для больших приложений Enterprise-уровня более предпочтительным является Dependency Injection.