



МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет Информационных технологий
Кафедра Информатики и информационных технологий

направление подготовки

09.03.02 «Информационные системы и технологии»

ЛАБОРАТОРНАЯ РАБОТА № 3

Дисциплина: «Backend»

Тема: *Создание консольного приложения с внедренными зависимостями на основе ASP.NET Core*

Выполнил: студент группы: 231-339

Карапетян Нвер Каренович

(Фамилия И.О.)

Дата, подпись: 21.02.25

(Дата)

(Подпись)

Проверил: _____

(Фамилия И.О., степень, звание)

(Оценка)

Дата, подпись _____

(Дата)

(Подпись)

Москва
2025

Цель:

Освоить создание консольного приложения, использующего внедрение зависимостей в ASP.NET Core.

Задачи:

1. Создать новый проект консольного приложения ASP.NET Core.
2. Настроить внедрение зависимостей для работы с различными сервисами или компонентами приложения.
3. Реализовать основные функции приложения, используя внедренные зависимости.
4. Протестировать работу приложения, обеспечив его функциональность в консольном режиме.

Ход работы

Проблема связанности и масштабируемости ПО

В процессе разработки программного обеспечения часто возникает проблема жесткой связанности компонентов. Когда один класс напрямую создает экземпляры других классов, он становится зависимым от конкретных реализаций, что затрудняет модификацию, тестирование, масштабируемость и повторное использование кода. Под **зависимостями** здесь понимается некоторая сущность, от которой зависит другая сущность. В качестве аналогии можно привести в пример различные строителя, который зависит от различных строительных инструментов для выполнения своей работы. Рассмотрим следующий экземпляр кода (рис. 1):

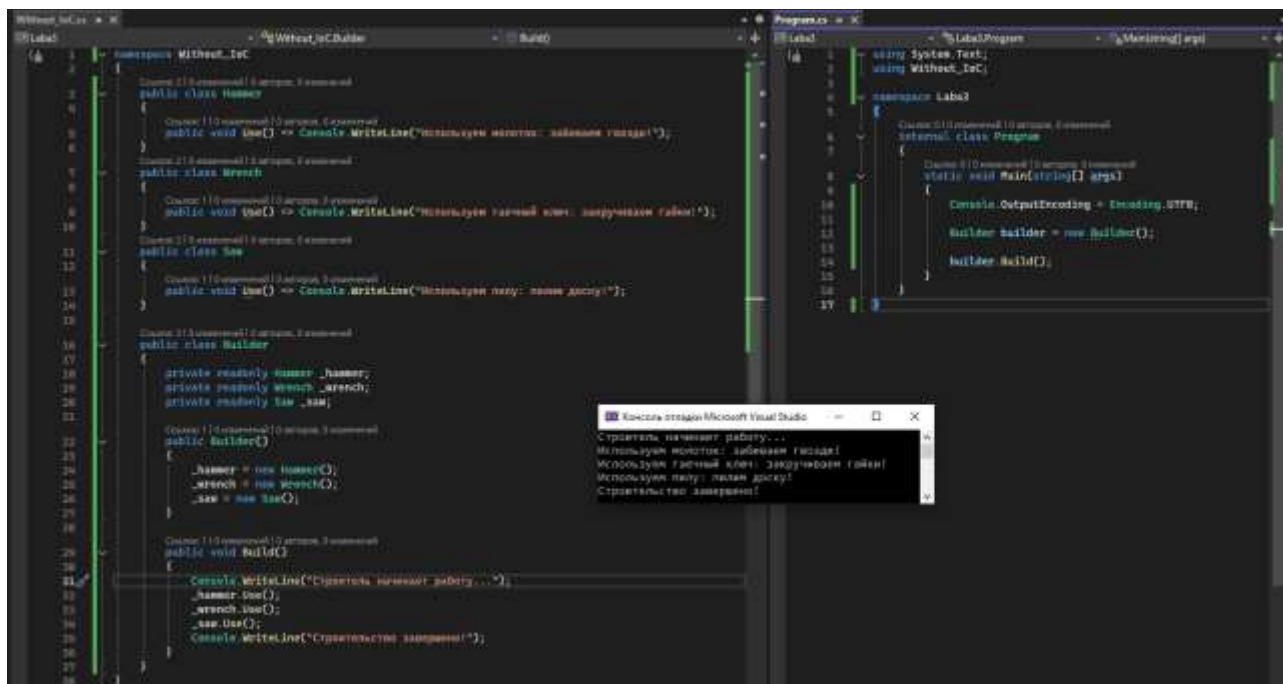


Рисунок 1. Листинг скрипта Without_IoC.cs

В приведенном скрипте класс строителя (Builder) вынужден создавать свои инструменты (Hammer, Wrench, Saw) самостоятельно (строки 24-26) и при необходимости замены одного инструмента на другой потребуются вносить изменения в сам класс строителя, что нарушает **принцип единственной ответственности (SRP)** и усложняет поддержку системы.

Для решения этой проблемы был разработан принцип **Inversion of Control (IoC)** — инверсия управления. Вместо того чтобы класс сам управлял своими зависимостями, это управление передается внешнему механизму, который решает, какие зависимости использовать. Реализация IoC может осуществляться разными способами, одним из которых является Dependency Injection (DI).

Работа с Dependency Injection

Dependency Injection (DI) — это конкретный механизм внедрения зависимостей, позволяющий передавать объекты в классы через конструкторы, сеттеры или интерфейсы. Этот подход обеспечивает гибкость и удобство работы с зависимостями, так как их создание и управление перекладывается на специальный контейнер, который разрешает зависимости и предоставляет нужные экземпляры объектов.

1) Реализация Dependency Injection через конструкторы.

В этом варианте зависимости передаются в Builder через конструктор. Это уменьшает связанность между классами и делает код более гибким:

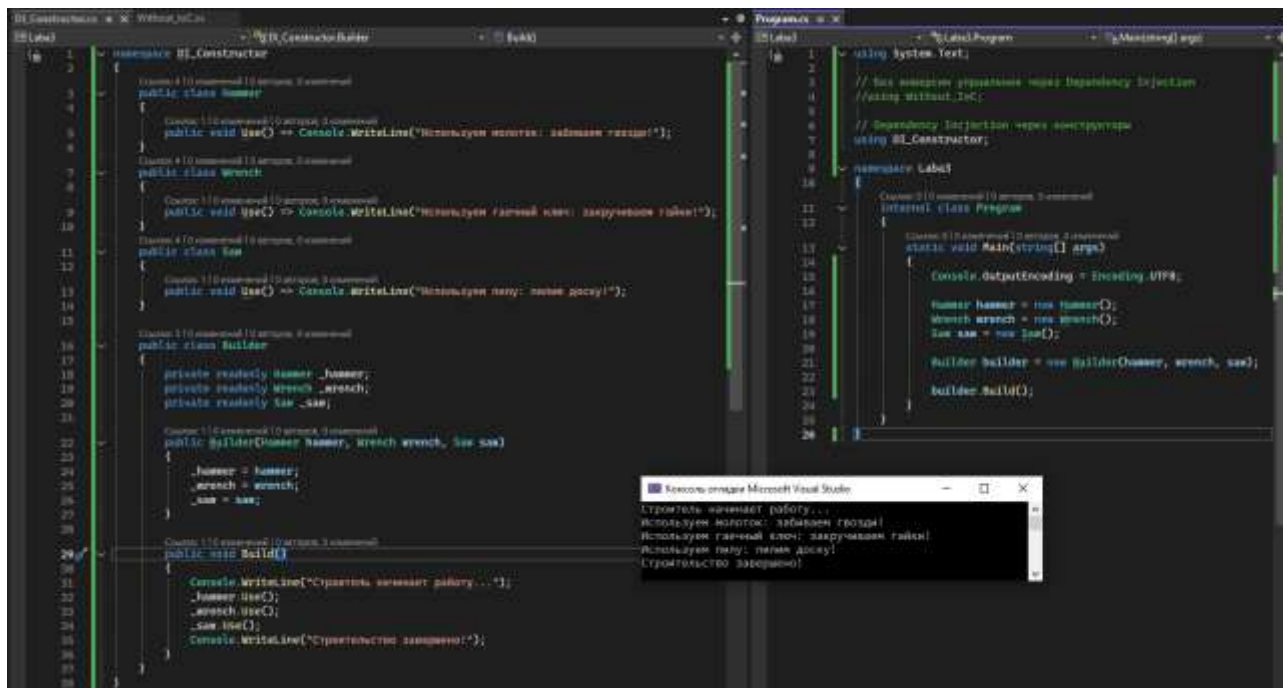


Рисунок 2. Листинг скрипта DI_Constructor.cs

Этот способ улучшает тестируемость, так как теперь можно передавать в Builder подставные реализации инструментов через параметры конструктора класса. Кроме того, код Builder теперь не зависит от конкретного способа создания инструментов, а также класс строителя больше не ответственен за создание экземпляров инструментов.

2) Реализация Dependency Injection через сеттеры.

Вариант внедрения зависимостей через сеттеры позволяет передавать зависимости через публичные свойства классов, что упрощает работу с объектами, однако может потребовать дополнительных проверок на null и не всегда может быть лучшим вариантом с точки зрения тестируемости. Несмотря на это, в некоторых случаях это может быть удобным способом, особенно когда мы предполагаем, что значения зависимостей в ходе работы программы могут изменяться. В таком случае сеттеры позволяют гибко работать с зависимостями:

```

1 namespace DI_Setter
2 {
3     // Класс 0 (0 строк, 0 строк)
4     public class Hammer
5     {
6         // Класс 1 (1 строк, 1 строк)
7         public void Use() => Console.WriteLine("Используем молоток: забиваем гвозди!");
8     }
9
10    // Класс 2 (1 строк, 1 строк)
11    public class Wrench
12    {
13        // Класс 3 (1 строк, 1 строк)
14        public void Use() => Console.WriteLine("Используем гаечный ключ: закручиваем гайки!");
15    }
16
17    // Класс 4 (1 строк, 1 строк)
18    public class Saw
19    {
20        // Класс 5 (1 строк, 1 строк)
21        public void Use() => Console.WriteLine("Используем пилу: пилим доску!");
22    }
23
24    // Класс 6 (1 строк, 1 строк)
25    public class Builder
26    {
27        // Класс 7 (1 строк, 1 строк)
28        public void SetHammer(Hammer hammer) { _hammer = hammer; }
29        // Класс 8 (1 строк, 1 строк)
30        public void SetWrench(Wrench wrench) { _wrench = wrench; }
31        // Класс 9 (1 строк, 1 строк)
32        public void SetSaw(Saw saw) { _saw = saw; }
33
34        // Класс 10 (1 строк, 1 строк)
35        public void Build()
36        {
37            if (_hammer is null || _wrench is null || _saw is null)
38            {
39                throw new Exception("Не все инструменты имеются у строителя...");
40            }
41
42            Console.WriteLine("Строитель начинает работу...");
43            _hammer.Use();
44            _wrench.Use();
45            _saw.Use();
46            Console.WriteLine("Строительство завершено!");
47        }
48    }
49 }

```

```

1 using System.Text;
2
3 // See https://aka.ms/aspnetcore-dependencyinjection for more information on this file.
4 //using Microsoft.Extensions.DependencyInjection;
5
6 // Dependency Injection через конструкторы
7 //using DI_Constructor;
8
9 // Dependency Injection через свойства
10 using DI_Setter;
11
12 namespace Labels
13 {
14     // Класс 1 (1 строк, 1 строк)
15     internal class Program
16     {
17         // Класс 2 (1 строк, 1 строк)
18         static void Main(string[] args)
19         {
20             Console.OutputEncoding = Encoding.UTF8;
21
22             Hammer hammer = new Hammer();
23             Wrench wrench = new Wrench();
24             Saw saw = new Saw();
25
26             Builder builder = new Builder();
27
28             // Dependency Inverted class через свойства класса
29             builder.Hammer = hammer;
30             builder.Wrench = wrench;
31             builder.Saw = saw;
32
33             builder.Build();
34         }
35     }
36 }

```

Консоль отладки Microsoft Visual Studio

```

Строитель начинает работу...
Используем молоток: забиваем гвозди!
Используем гаечный ключ: закручиваем гайки!
Используем пилу: пилим доску!
Строительство завершено!

```

Рисунок 3. Листинг скрипта DI_Setter.cs

3) Реализация Dependency Injection через интерфейсы.

```

1 namespace DI_Interface
2 {
3     // Класс 0 (1 строк, 1 строк)
4     public interface ITool
5     {
6         // Класс 1 (1 строк, 1 строк)
7         public void Use();
8     }
9
10    // Класс 2 (1 строк, 1 строк)
11    public class Hammer : ITool
12    {
13        // Класс 3 (1 строк, 1 строк)
14        public void Use() => Console.WriteLine("Используем молоток: забиваем гвозди!");
15    }
16
17    // Класс 4 (1 строк, 1 строк)
18    public class Wrench : ITool
19    {
20        // Класс 5 (1 строк, 1 строк)
21        public void Use() => Console.WriteLine("Используем гаечный ключ: закручиваем гайки!");
22    }
23
24    // Класс 6 (1 строк, 1 строк)
25    public class Saw : ITool
26    {
27        // Класс 7 (1 строк, 1 строк)
28        public void Use() => Console.WriteLine("Используем пилу: пилим доску!");
29    }
30
31    // Класс 8 (1 строк, 1 строк)
32    public class Builder
33    {
34        // Класс 9 (1 строк, 1 строк)
35        private readonly ITool _hammer;
36        // Класс 10 (1 строк, 1 строк)
37        private readonly ITool _wrench;
38        // Класс 11 (1 строк, 1 строк)
39        private readonly ITool _saw;
40
41        // Класс 12 (1 строк, 1 строк)
42        public Builder(ITool hammer, ITool wrench, ITool saw)
43        {
44            _hammer = hammer;
45            _wrench = wrench;
46            _saw = saw;
47        }
48
49        // Класс 13 (1 строк, 1 строк)
50        public void Build()
51        {
52            Console.WriteLine("Строитель начинает работу...");
53            _hammer.Use();
54            _wrench.Use();
55            _saw.Use();
56            Console.WriteLine("Строительство завершено!");
57        }
58    }
59 }

```

```

1 using System.Text;
2
3 // See https://aka.ms/aspnetcore-dependencyinjection for more information on this file.
4 //using Microsoft.Extensions.DependencyInjection;
5
6 // Dependency Injection через конструкторы
7 //using DI_Constructor;
8
9 // Dependency Injection через свойства
10 using DI_Setter;
11
12 // Dependency Injection через интерфейс
13 using DI_Interface;
14
15 namespace Labels
16 {
17     // Класс 1 (1 строк, 1 строк)
18     internal class Program
19     {
20         // Класс 2 (1 строк, 1 строк)
21         static void Main(string[] args)
22         {
23             Console.OutputEncoding = Encoding.UTF8;
24
25             Hammer hammer = new Hammer();
26             Wrench wrench = new Wrench();
27             Saw saw = new Saw();
28
29             Builder builder = new Builder(hammer, wrench, saw);
30
31             builder.Build();
32         }
33     }
34 }

```

Консоль отладки Microsoft Visual Studio

```

Строитель начинает работу...
Используем молоток: забиваем гвозди!
Используем гаечный ключ: закручиваем гайки!
Используем пилу: пилим доску!
Строительство завершено!

```

Рисунок 4. Листинг скрипта DI_Interface.cs

Внедрение зависимостей через интерфейсы обеспечит большую гибкость и позволяет использовать полиморфизм, что полезно для более сложных сценариев.

риев, где необходимо подменять зависимости в зависимости от контекста выполнения. К примеру, если нам понадобится вместо экземпляра обычной пилы (Saw) использовать более специфичный дочерний класс электрической пилы (ElectricSaw).

Реализация DI через библиотеку **DependencyInjection**

В предыдущем разделе мы рассматривали способы ручной реализации внедрения зависимостей. Такой подход дает полный контроль над зависимостями, однако требует дополнительных усилий для управления их жизненным циклом и созданием экземпляров. В отличие от ручной реализации, автоматическое внедрение зависимостей с использованием библиотеки **Microsoft.Extensions.DependencyInjection** значительно упрощает этот процесс и делает код более расширяемым и гибким.

В экосистеме ASP.NET Core для внедрения зависимостей используется стандартный DI-контейнер, который автоматизирует создание и управление объектами зависимостей. Эта библиотека предоставляет средства для регистрации сервисов в контейнере и управления их жизненным циклом, что позволяет улучшить масштабируемость и удобство разработки.

Библиотека **Microsoft.Extensions.DependencyInjection** является основой для работы с DI в .NET Core и уже встроена в проекты, созданные с использованием шаблонов ASP.NET Core. Однако, для других типов проектов необходимо добавить ее вручную. Это можно сделать через менеджер пакетов NuGet.

ServiceCollection — это контейнер сервисов, куда мы добавляем все зависимости. Мы регистрируем наши инструменты (Hammer, Wrench, Saw) и их зависимости через метод **AddTransient**, который означает, что для каждого запроса будет создаваться новый экземпляр объекта.

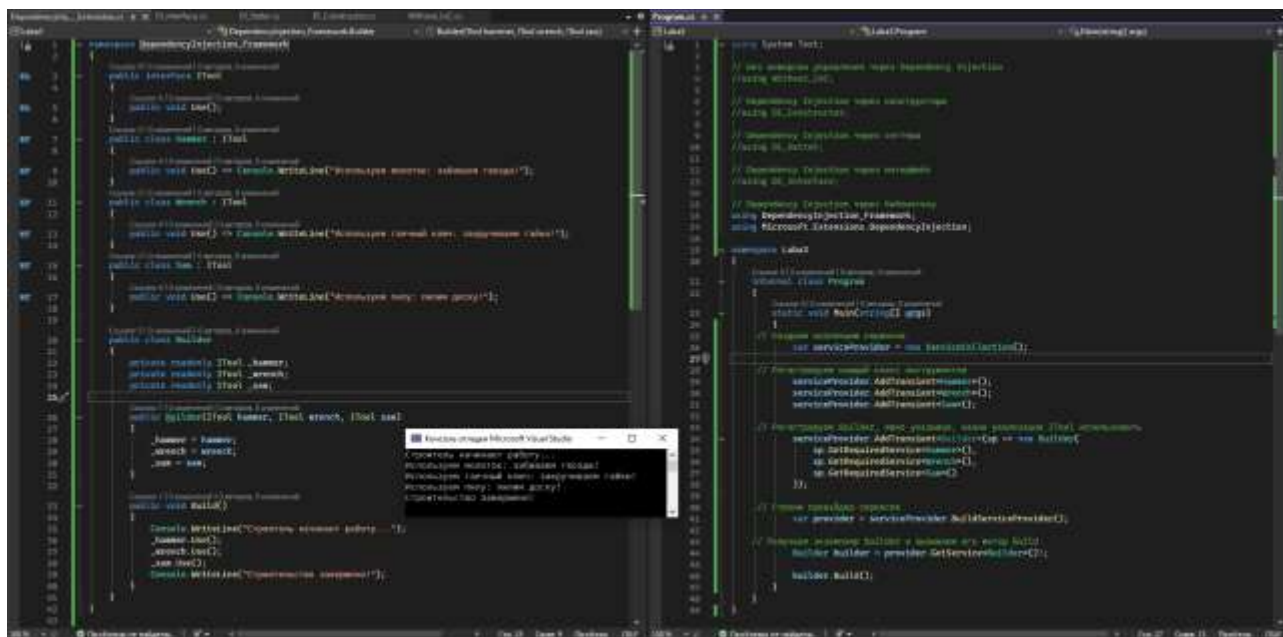


Рисунок 5. Листинг скрипта `DependencyInjection_Framework.cs`

Реализация **Builder** — мы явным образом передаем зависимости в конструктор класса `Builder` через `AddTransient` и используем делегат для того, чтобы в конструктор передавались конкретные реализации инструментов (`Hammer`, `Wrench`, `Saw`).

BuildServiceProvider — строим сервис-провайдер, который будет управлять созданием объектов.

Затем, воспользовавшись методом **GetService**, мы сможем получить экземпляр `Builder`, в который уже внедрены все необходимые зависимости. Следующим шагом вызываем метод `Build` для выполнения основной логики приложения, где используются инструменты.

Заключение

В ходе выполнения лабораторной работы был освоен механизм инверсии управления (IoC), в частности один из способов его реализации — `Dependency Injection`. Были написаны и разобраны скрипты как без внедрения зависимостей, так и с внедрением зависимостей вручную (через конструкторы, сеттеры и интерфейсы) и автоматически (с помощью библиотеки `.NET DI`).