ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

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):

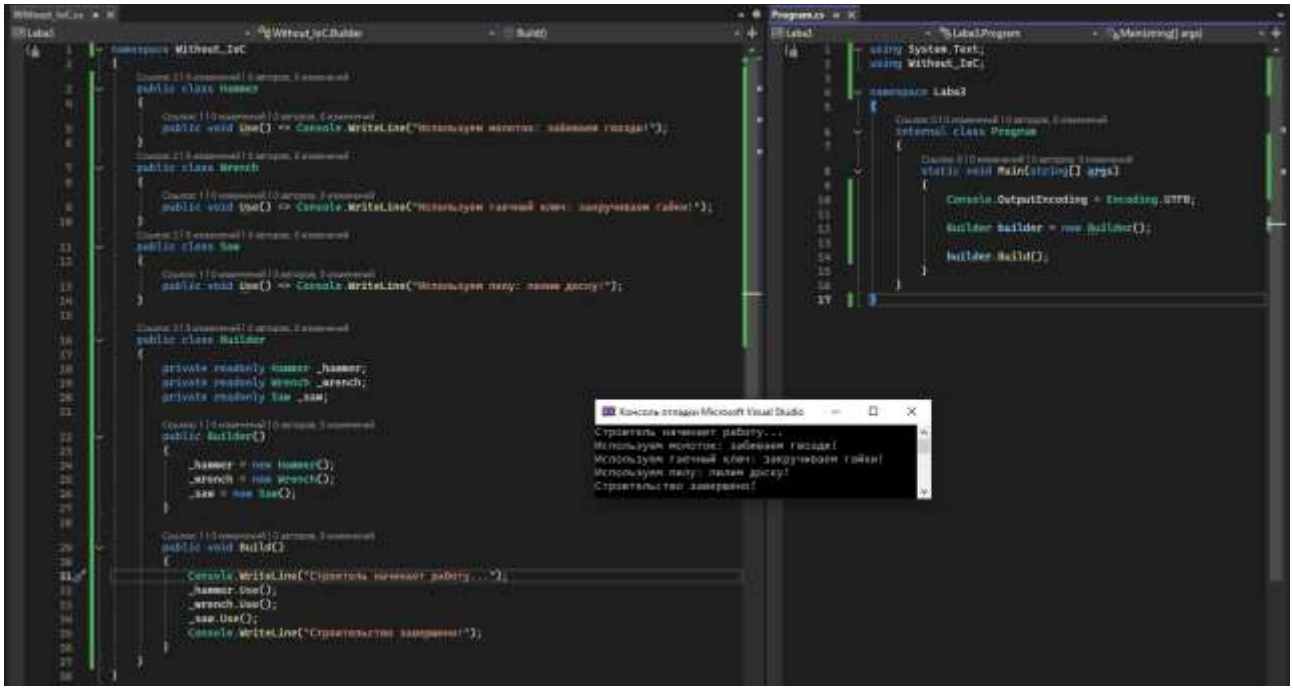

Рисунок 1. Листинг скрипта Without_IoC.cs

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 потребуется 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:

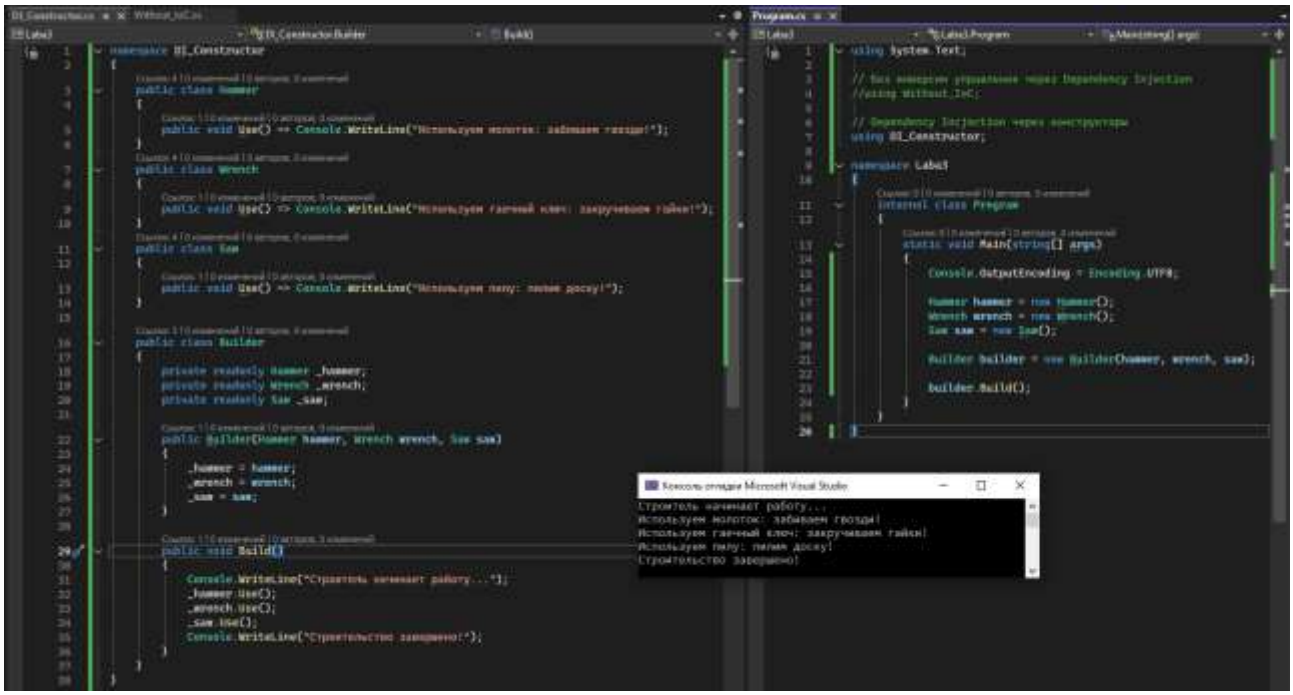

Рисунок 2. Листинг скрипта DI_Constructor.cs

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 позволят гибко работать с зависимостями:

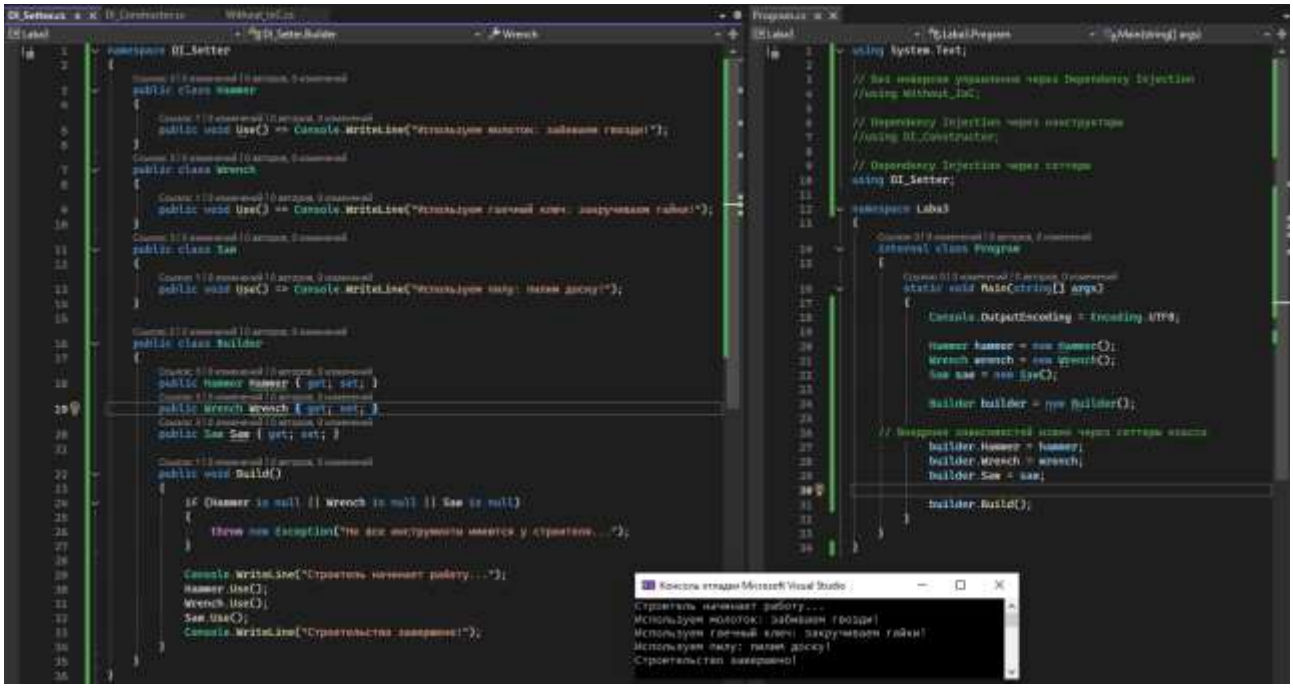

Рисунок 3. Листинг скрипта DI_Setter.cs

3) **Реализация Dependency Injection через интерфейсы.**

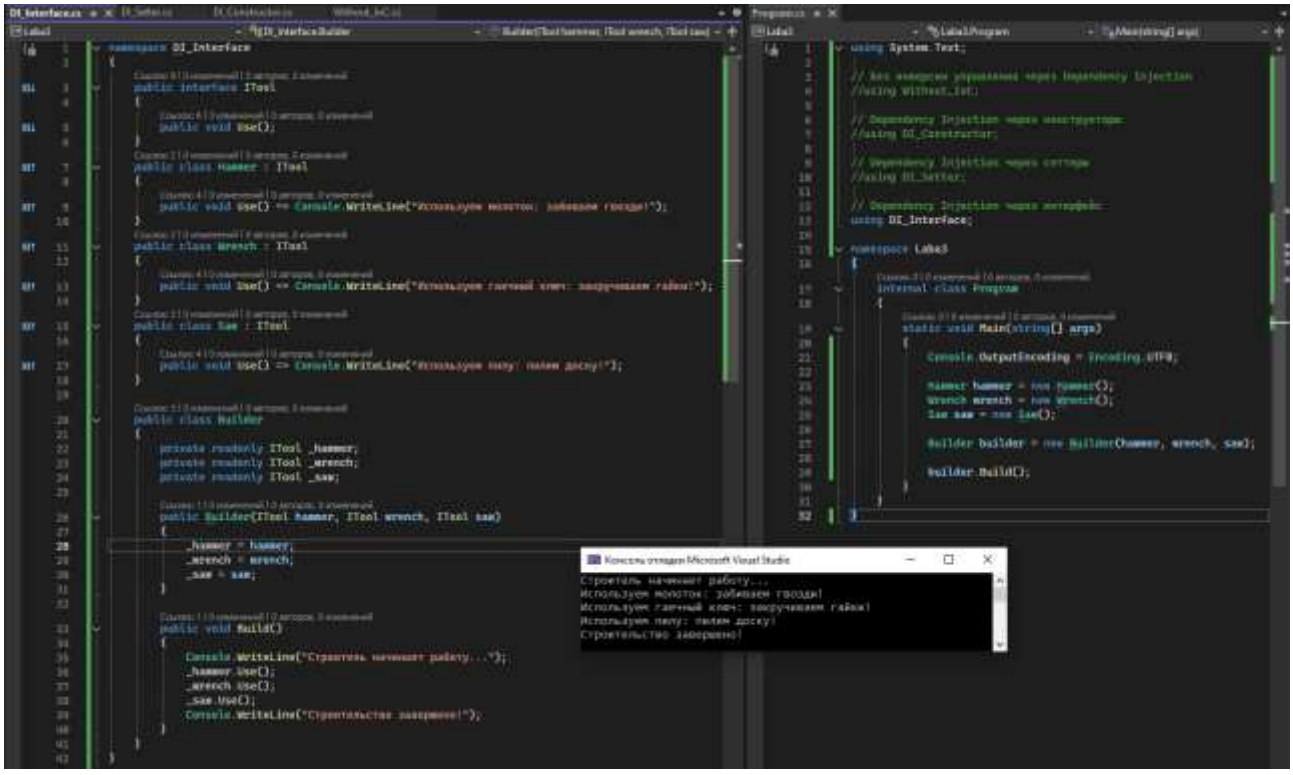

Рисунок 4. Листинг скрипта DI_Interface.cs

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

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.

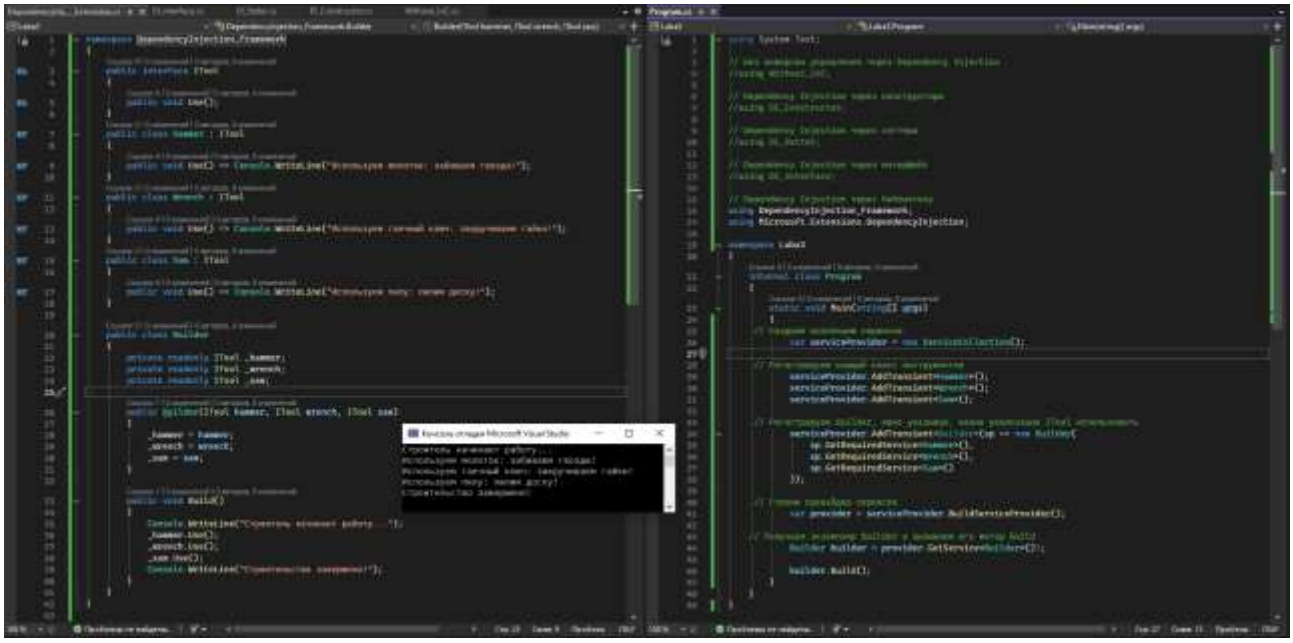

Рисунок 5. Листинг скрипта DependencyInjection_Framework.cs

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).