Занятие 3. IoC

Сложность разрабатываемой информационной системы увеличивается, в связи с чем вы решаете начать автоматизацию процесса внедрения зависимостей.

Цель занятия

1. Получить практический опыт организации автоматического внедрения зависимостей посредством паттерна Service Locator.
2. Получить практический опыт организации автоматического внедрения зависимостей посредством DI.

Использование паттерна Service Locator

В качестве контейнера зависимостей будем использовать стандартный инструмент от Microsoft - пакет `Microsoft.Extensions.DependencyInjection`.

Для его добавления в проект можно использовать как UI-средства, предоставляемые IDE, так и обычный терминал. Рассмотрим способ с терминалом:

1. Перейти в каталог с проектом
2. Запустить терминал
3. В терминале выполнить команду `dotnet add package Microsoft.Extensions.DependencyInjection`
4. После скачивания пакета он будет добавлен в проект

При организации внедрения зависимостей через данный пакет работа делится на следующие этапы:

1. Создание коллекции зависимостей
2. Регистрация зависимостей в коллекции
3. Построение контейнера зависимостей
4. Разрешение зависимостей

Создание коллекции зависимостей происходит путем создания объекта `Microsoft.Extensions.DependencyInjection.ServiceCollection`:

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();
```

В качестве сервисов зарегистрируем классы `CarService`, `CustomerStorage`, `PedalCarFactory` и `HandCarFactory`, при этом убрав старый код, который мы использованли для создания экземпляров данных классов.

Так как каждый сервис из перечисленных существует лишь в единственном экземпляре в нашем приложении, в качестве времени жизни будем использовать `Singleton`.

В итоге `Program.cs` будет начинаться со следующего кода:

```
using Microsoft.Extensions.DependencyInjection;
using UniversalCarShop;

var services = new ServiceCollection();

services.AddSingleton<CarService>();
services.AddSingleton<CustomersStorage>();
services.AddSingleton<PedalCarFactory>();
services.AddSingleton<HandCarFactory>();
```

Сейчас в нашем контейнере зависимостей есть лишь классы, но мы помним, что `HseCarService` использует интерфейсы `ICarProvider` и `ICustomersProvider`, а не конкретные реализации. Чтобы в нашем контейнере зависимостей стали доступны данные интерфейсы, их также нужно зарегистрировать. Чтобы зарегистрировать в качестве сервиса интерфейс, реализуемый классом `CarService`, можно использовать следующий код:

```
services.AddSingleton<ICarProvider>(sp => sp.GetRequiredService<CarService>());
```

Аналогично для класса `CustomersStorage`:

```
services.AddSingleton<ICustomersProvider>(sp =>
sp.GetRequiredService<CustomersStorage>());
```

Когда зависимости зарегистрированы, необходимо построить контейнер. Происходит это путем вызова метода `BuildServiceProvider` класса `ServiceCollection`:

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

Чтобы `HseCarService` мог использовать контейнер, его необходимо настроить:

1. В конструкторе класса заменить все параметры на единственный параметр `IServiceProvider`:
2. Используя метод `GetRequiredService` данного интерфейса, получить зависимости `ICarProvider` и `ICustomersProvider`

Код конструктора должен стать следующим:

```
public HseCarService(IServiceProvider serviceProvider)
{
```

```
    _carProvider = serviceProvider.GetRequiredService<ICarProvider>();
    _customersProvider = serviceProvider.GetRequiredService<ICustomersProvider>();
}
```

Теперь можно построить экземпляр данного класса:

```
var hse = new HseCarService(serviceProvider);
```

Также получим экезмпляры остальных наших сервисов - теперь уже из контейнера зависимостей:

```
var customers = serviceProvider.GetRequiredService<CustomersStorage>();
var cars = serviceProvider.GetRequiredService<CarService>();
var pedalCarFactory = serviceProvider.GetRequiredService<PedalCarFactory>();
var handCarFactory = serviceProvider.GetRequiredService<HandCarFactory>();
```

Запустим программу и убедимся, что ничего не сломалось.

В данном случае мы применили паттерн Service Locator по отношению к классу `HseCarService`.

Замена Service Locator на Dependency Injection

Паттерн Service Locator полезен в некоторых случаях, но у него есть недостатки:

1. Класс, в который внедряются зависимости, знает о контейнере зависимостей - это в свою очередь ухудшает портируемость кода
2. По контрактам, которые класс предоставлят, нельзя понять, какие именно зависимости он использует - из-за этого ухудшается тестируемость кода

Чтобы избавиться от данных недостатков, применяется паттерн Dependency Injection. Реализовать его можно путем внесения небольших изменений в уже написанный нами код.

Для начала вернем в конструктор класса `HseCarService` зависимость от интерфейсов `ICarProvider` и `ICustomersProvider`:

```
public HseCarService(ICarProvider carProvider, ICustomersProvider
customersProvider)
{
    _carProvider = carProvider;
    _customersProvider = customersProvider;
}
```

Далее необходимо зарегистрировать класс в качестве сервиса в контейнере зависимостей:

```
services.AddSingleton<HseCarService>();
```

Теперь можно просто запросить экземпляр данного класса из контейнера так же, как и остальные сервисы:

```
var hse = serviceProvider.GetRequiredService<HseCarService>();
```

Запускаем код и проверяем, что ничего не сломалось.

Как мы можем видеть, теперь класс `HseCarService` зависит только от нашего собственного кода и не опирается на код программной инфраструктуры, что, повышает его портируемость и тестируемость.