

Лабораторная работа №6.

Внедрение зависимости с помощью IoC контейнера

Цель работы

Приобретение практических навыков использования IoC контейнера Castle Windsor для внедрения зависимости; практика использования шаблонов проектирования; практика использования тестового каркаса NUnit, практика использования изолирующего каркаса NSubstitute.

Задание на лабораторную работу

1. Подготовить учебный проект
2. Подключить в проект пакет Castle Windsor
3. Реализовать паттерна команда
4. Добавить и сконфигурировать IoC контейнера
5. Добавить декоратор для команд
6. Добавить декоратора для перехвата исключений
7. Реализовать автономные тесты для разработанных классов
8. На каждом шаге делайте снимки исходного кода создаваемых или изменяемых классов и тестов, окна «Результаты тестов» и «Обозреватель решения» и сохраните в документе MS Word.
9. Оформить отчет

Порядок выполнения работы

1. Подготовка проекта

Для выполнения данной лабораторной работы возьмите решение, полученное в результате выполнения лабораторной работы №5.

Выполните тесты.

Зафиксируйте исходное состояние окна «Обозреватель решения», окна «Результаты тестов»: и сохраните в документе MS Word.

2. Подключить в проект «.Lib» IoC контейнер Castle Windsor

1. Открыть Диспетчер пакетов Nuget

2. Источник пакета указать «nuget.org»
3. Переключиться на вкладку Обзор и набрать в строке Поиск «NSubstitute».
4. Необходимо подключить пакет **Castle Windsor**. Отметить его использование в проекте «.Lib»



3. Реализация паттерна команда

1. В проекте «.Lib» создайте папку «src/SampleCommands»
2. Создайте в ней интерфейс соответствующий шаблону команда:

```
public interface ISampleCommand
{
    // Ссылка: 5
    void Execute();
}
```

3. Добавьте в этой же папке два класса, реализующие этот интерфейс: FirstCommand и SecondCommand

```
public class FirstCommand : ISampleCommand
{
    // Ссылка: 3
    public void Execute()
    {
        throw new System.NotImplementedException();
    }
}
```

4. Пусть команда должна уметь выводить имя своего класса, и количество раз вызова метода Execute() для экземпляра класса. Вывод будем осуществлять с помощью интерфейса IView, созданного в предыдущей лабораторной работе. Будем использовать внедрение зависимости через конструктор. Для подсчета количества вызовов организуем поле — счетчик.

```

Ссылка: 4
public class FirstCommand : ISampleCommand
{
    Ссылка: 0
    public FirstCommand(IView view)
    {
        this.view = view;
    }

    private readonly IView view;

    private int iExecute = 0;

    Ссылка: 3
    public void Execute()
    {
        throw new System.NotImplementedException();
    }
}

```

5. Добавьте реализацию метода Execute, добавить приращение счетчика и вызов метода IView.Render():

```
view.Render(this.GetType().ToString() + "\n iExecute = " + iExecute);
```

6. Аналогичным образом реализуйте SecondCommand.
7. В проекте «.Service» создайте папку «src\Views»
8. В этой папке создайте класс ConsoleView, реализующий интерфейс View:

```

class ConsoleView : IView
{
    Ссылка: 7
    public void Render(string text)
    {
        Console.WriteLine(text);
    }
}

```

9. Зафиксируйте в отчете код классов и состояние окна Обозреватель решения.

4. Добавление и конфигурация контейнера

1. В проекте «.Lib» создайте папку «src/Common»
2. Создайте в ней статический класс CastleFactory. В нем разметим ссылку на IoC контейнер.

```

Ссылка: 3
public static class CastleFactory
{
    /// <summary>Контейнер </summary>
    Ссылка: 3
    public static IWindsorContainer container { get; private set; }

    Ссылка: 0
    static CastleFactory()
    {
        //создать объект контейнер
        container = new WindsorContainer();
    }
}

```

3. В проекте «.Service» создайте папку «src/WindsorInstallers»

4. Добавьте в нее класс для конфигурирования «команд»

```
class SampleCommandInstaller : IWindsorInstaller
{
    Ссылка: 0
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(
            Component.For<ISampleCommand>().ImplementedBy<FirstCommand>().LifeStyle.Transient
        );
    }
}
```

Жизненный цикл объектов укажем Transient – при каждом разрешении зависимости будет создаваться новый экземпляр объекта

5. Добавьте в папку «src/WindsorInstallers» класс для конфигурирования «представления» ViewInstaller:

```
class ViewInstaller : IWindsorInstaller
{
    Ссылка: 0
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(
            Component.For<IView>().ImplementedBy<ConsoleView>().LifeStyle.Transient);
    }
}
```

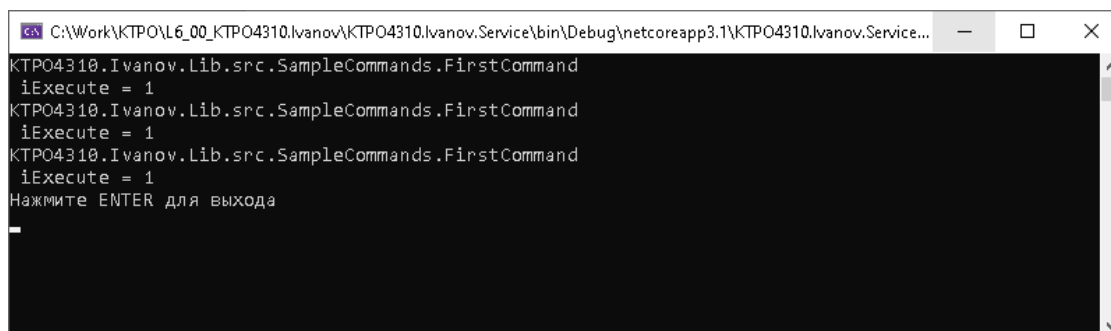
6. Добавьте в класс «Program» конфигурацию контейнера в начало метода main:

```
CastleFactory.container.Install(
    new SampleCommandInstaller(),
    new ViewInstaller()
);
```

7. Добавьте в класс «Program» код вызова «команд»:

```
for (int i = 0; i < 3; i++)
{
    ISampleCommand someCommand = CastleFactory.container.Resolve<ISampleCommand>();
    someCommand.Execute();
}
```

8. Соберите решение и запустите проект «.Service». Ожидаемый результат:



Обратите внимание на значение счетчика: он равен 1 для всех обращений. Почему?

9. Зафиксируйте в отчете код классов, результат запуска и состояние окна
Обозреватель решения.

10. Заменить в регистрации зависимости для команды жизненный цикл объекта на Singleton. Запустите проект «.Service». Значение счетчика изменяется. Почему?

Зафиксируйте в отчете код измененных классов и результат запуска.

11. Заменить в регистрации зависимости для класса FirstCommand на SecondCommand. Запустите проект «.Service».

Зафиксируйте в отчете код измененных классов и результат запуска.

5. Добавление декоратора

1. Создайте в папке «SampleCommands» проект «.Lib» класс SampleCommandDecorator реализующий интерфейс ISampleCommand.
2. В конструкторе класса в качестве параметра передайте объект класса ISampleCommand, определите и инициализируйте поле для него.
3. Добавьте в метод Execute() вызов метода объекта переданного в конструкторе. Это класс будет соответствовать паттерну декоратор:

```
public class SampleCommandDecorator : ISampleCommand
{
    private readonly ISampleCommand sampleCommand;

    Ссылка: 0
    public SampleCommandDecorator(ISampleCommand sampleCommand)
    {
        this.sampleCommand = sampleCommand;
    }

    Ссылка: 3
    public void Execute()
    {
        sampleCommand.Execute();
    }
}
```

4. Добавьте функциональность декоратора: пусть он выводит текст «Начало» до вызова декорируемого объекта, и «Конец» - после. Нужно в него внедрить через конструктор представления IView. Метод Execute() декоратора :

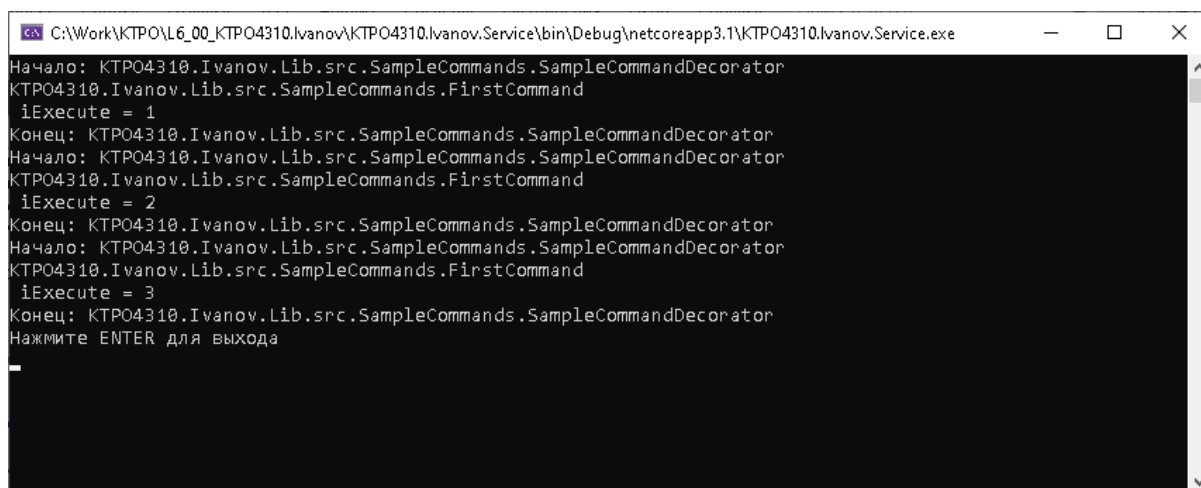
```
public void Execute()
{
    view.Render("Начало: " + this.GetType().ToString());

    try
    {
        sampleCommand.Execute();
    }
    finally
    {
        view.Render("Конец: " + this.GetType().ToString());
    }
}
```

5. Сконфигурируйте зависимости команд, добавив регистрацию декоратора:

```
public void Install(IWindsorContainer container, IConfigurationStore store)
{
    container.Register(
        Component.For<ISampleCommand>().ImplementedBy<SampleCommandDecorator>().LifeStyle.Singleton,
        Component.For<ISampleCommand>().ImplementedBy<FirstCommand>().LifeStyle.Singleton
    );
}
```

6. Соберите решение и запустите проект «.Service». Ожидаемый результат:



```
C:\Work\КТПО\Л6_00_КТПО4310.Ivanov\КТПО4310.Ivanov.Service\bin\Debug\netcoreapp3.1\КТПО4310.Ivanov.Service.exe
Начало: КТПО4310.Ivanov.Lib.src.SampleCommands.SampleCommandDecorator
КТПО4310.Ivanov.Lib.src.SampleCommands.FirstCommand
iExecute = 1
Конец: КТПО4310.Ivanov.Lib.src.SampleCommands.SampleCommandDecorator
Начало: КТПО4310.Ivanov.Lib.src.SampleCommands.SampleCommandDecorator
КТПО4310.Ivanov.Lib.src.SampleCommands.FirstCommand
iExecute = 2
Конец: КТПО4310.Ivanov.Lib.src.SampleCommands.SampleCommandDecorator
Начало: КТПО4310.Ivanov.Lib.src.SampleCommands.SampleCommandDecorator
КТПО4310.Ivanov.Lib.src.SampleCommands.FirstCommand
iExecute = 3
Конец: КТПО4310.Ivanov.Lib.src.SampleCommands.SampleCommandDecorator
Нажмите ENTER для выхода
```

7. Зафиксируйте в отчете код классов, результат запуска и состояние окна Обозреватель решения.

6. Добавление декоратора для перехвата исключений

1. Самостоятельно реализуйте еще один «декоратор», задаче которого будет перехват исключений из декорируемого объекта и вывод текста сообщения на экран. По порядку вызова он должен быть после `SampleCommandDecorator`.
2. Проведите эксперимент: добавьте в декорируемую команду `SecondCommand` вызов исключения. Сконфигурируйте зависимости, чтобы использовался этот класс.
3. Соберите решение и запустите проект «.Service». Зафиксируйте окно результат выполнения программы.
4. Зафиксируйте в отчете код классов, результат запуска и состояние окна Обозреватель решения.

7. Реализация автономных тестов для разработанных классов

Используя знания полученные на предыдущих лабораторных работах самостоятельно реализуйте автономные тесты для новых созданных классов. Для создания подделок используйте изолирующий каркас. Соблюдайте соглашения о именовании и размещении тестов.

1. Автономный тест для «команды» FirstCommand: метод Execute() вызывает вывод текста согласно заданию.
2. Автономный тест для «декоратора» SampleCommandDecorator: метод Execute() вызывает метод декорируемого объекта.
3. Автономный тест для «декоратора» SampleCommandDecorator: метод Execute() вызывает вывод текста согласно заданию.
4. Автономный тест для «декоратора» ExceptionCommandDecorator: метод Execute() вызывает метод декорируемого объекта.
5. Автономный тест для «декоратора» SampleCommandDecorator: метод Execute() обрабатывает исключения, возникшие в декорируемом объекте.

Зафиксируйте в отчете код тестов, код измененных классов, результаты выполнения тестов - окна «Результаты тестов», окно «Обозреватель решения».

Содержание отчета

1. Постановка задачи.
2. Экранные формы с результатами выполнения задания: окна «Обозреватель решения», окна «Обозреватель тестов», исходный код тестов, исходный код тестируемых классов и поддельных объектов.
3. Выводы.

Литература

1. Внедрение зависимостей в .NET, глава 10.1. Знакомство с Castle Windsor, <https://smarly.net/dependency-injection-in-net/di-containers/castle-windsor/introducing-castle-windsor>
2. Внедрение зависимостей в .NET, глава 10.2 Управление жизненным циклом <https://smarly.net/dependency-injection-in-net/di-containers/castle-windsor/managing-lifetime>
3. IoC-контейнер Castle Windsor, <https://metanit.com/sharp/mvc5/21.5.php>