**Лабораторная работа №6**

**Внедрение зависимости с помощью IoC контейнера**

**Цель работы:**

Приобретение практических навыков использования IoC контейнера Castle

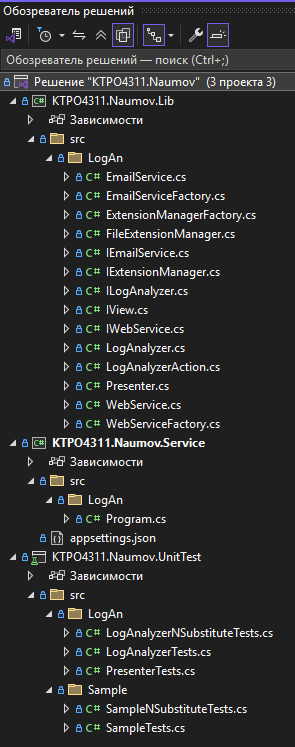
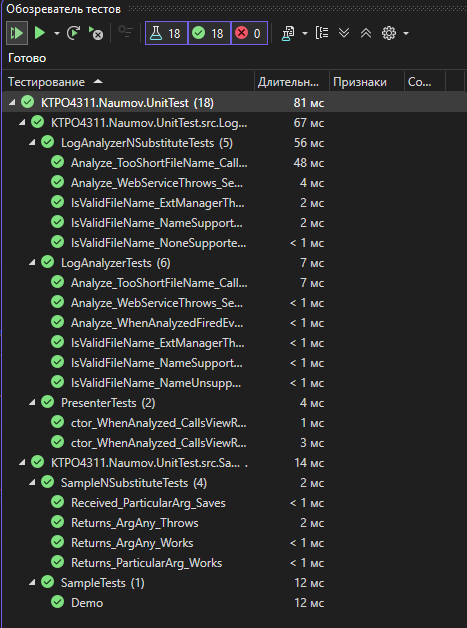
Windsor для внедрения зависимости; практика использования шаблонов

проектирования; практика использования тестового каркаса NUnit, практика

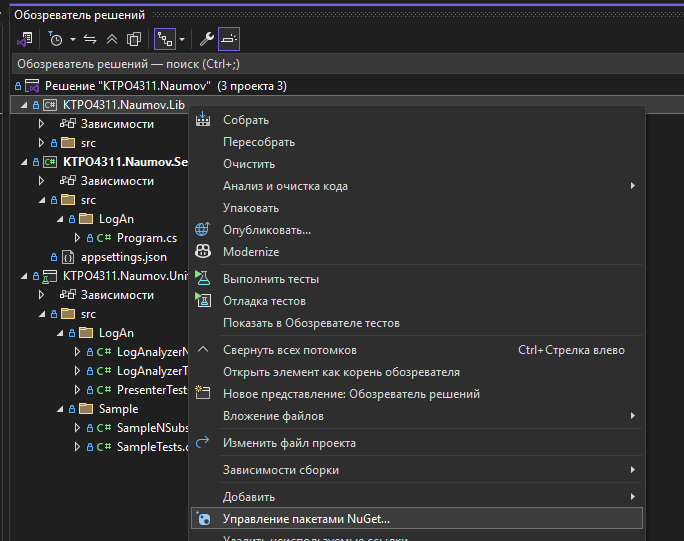
использования изолирующего каркаса NSubstitute.

**Порядок выполнения работы**

1. **Подготовка проекта**
2. Берём решение, полученное в результате выполнения лабораторной работы № 5
3. Выполним тесты.
4. Фиксируем исходное состояние проекта.



1. **Подключить в проект «.Lib» IoC контейнер Castle Windsor**

****

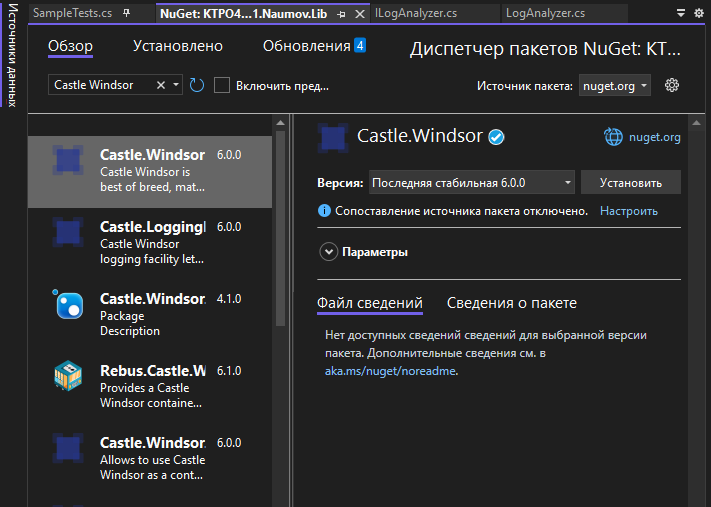
**4**

* 

**ПКМ**

**2**

**1**

****

**6**

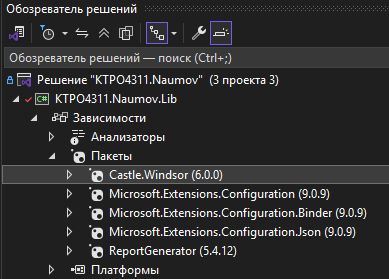
* 

**5**

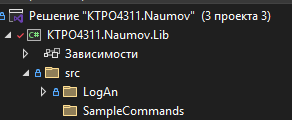
* 

**3**

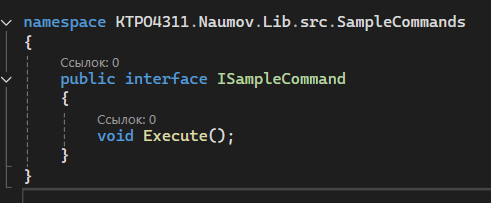
* 

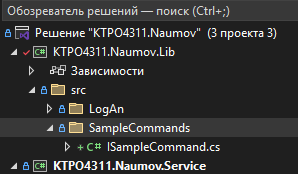
****

1. **Реализация паттерна команда.**
2. В проекте .Lib создаём папку src/SampleCommands

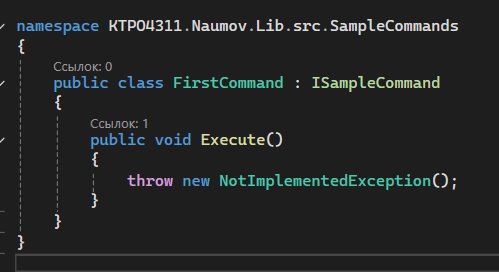
****

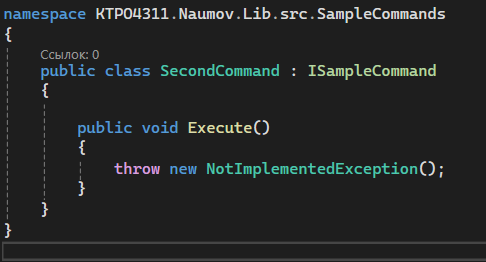
1. Создаём в новой папке интерфейс ISampleCommand.

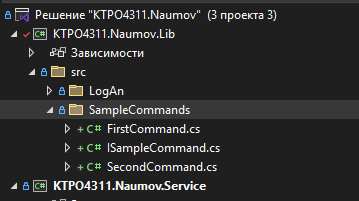




1. Создадим в этой же папке два класса, реализующие интерфейс







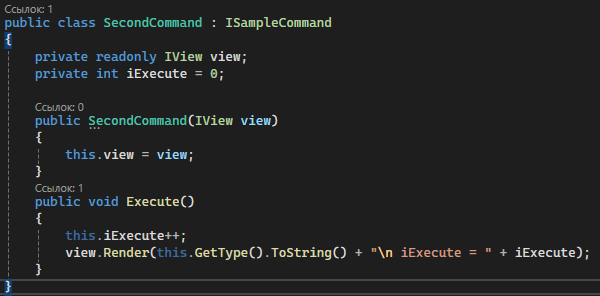
1. Создаём поля и конструктор.



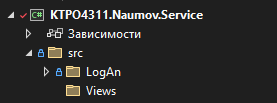
1. Добавляем реализацию методу Execute() – приращение счётчика и вызов метода IView.Render().



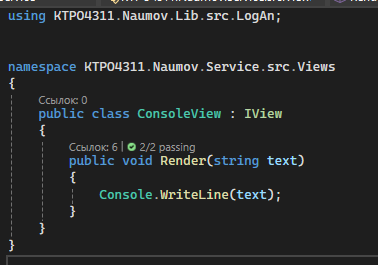
1. Аналогичным образом реализуем SecondCommand.



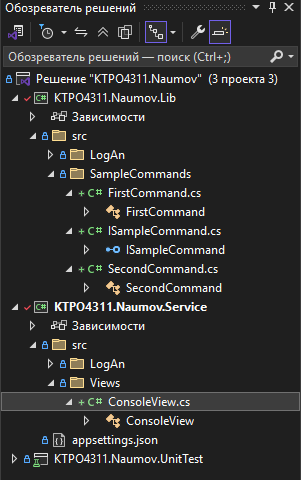
1. Создаём в проекте .Service папку src/Views



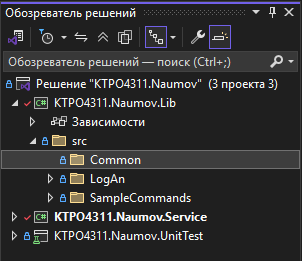
1. Создаём в этой папке класс ConsoleView, реализующий интерфейс IView



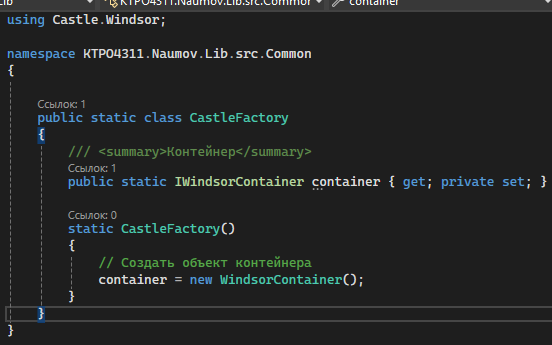
1. Фиксируем состояние обозревателя решений

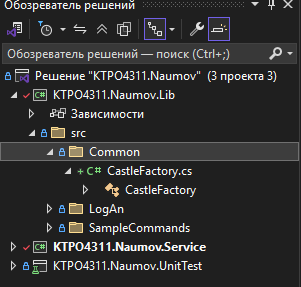


1. **Добавление и конфигурация контейнера.**
2. В проекте .Lib создаём папку src/Common.

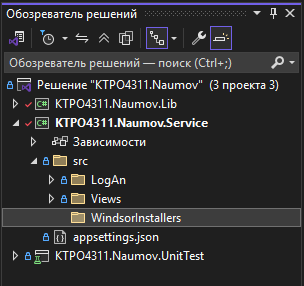


1. Создаём в новой папке статический класс CastleFactory. В нём разместим ссылку на IoC контейнер.

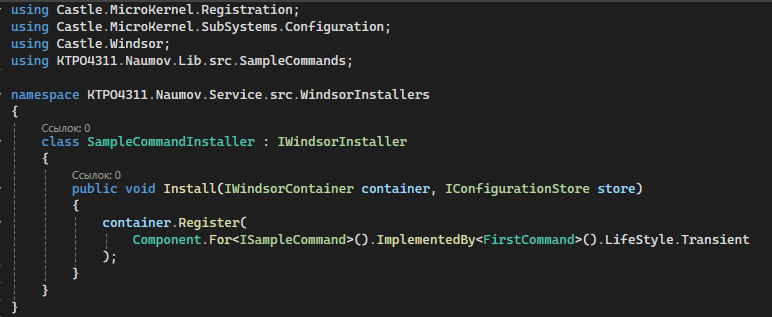


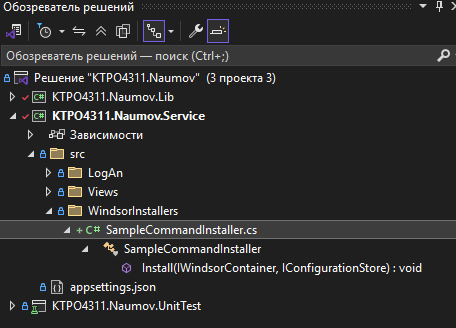


1. В проекте .Service создаём папку src/WindsorInstallers.



1. Добавим в неё класс для конфигурирования команд.



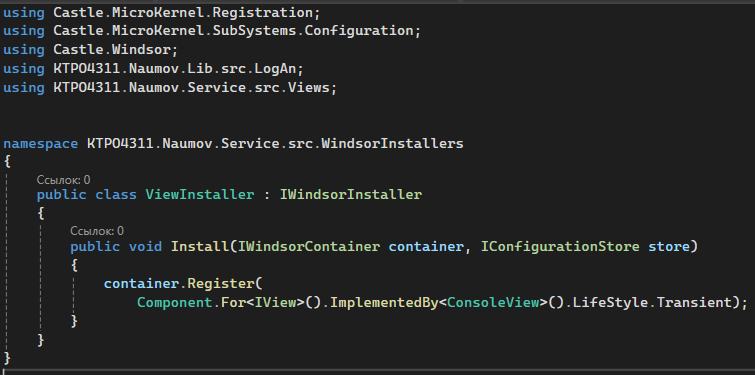


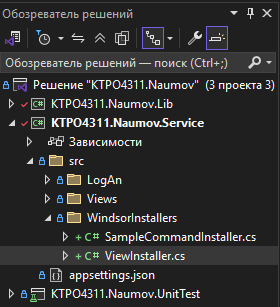
Жизненный цикл объектов укажем Transient – при каждом разрешении

зависимости будет создаваться новый экземпляр объекта.

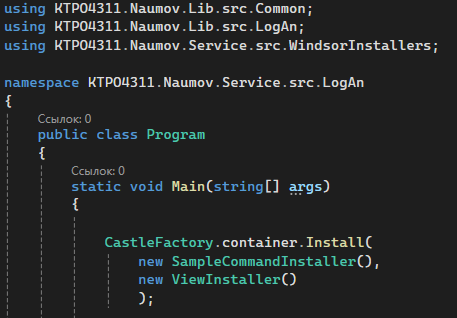
1. Добавим в папку src/WindsorInstallers класс для конфигурирования

“представления” ViewInstaller.

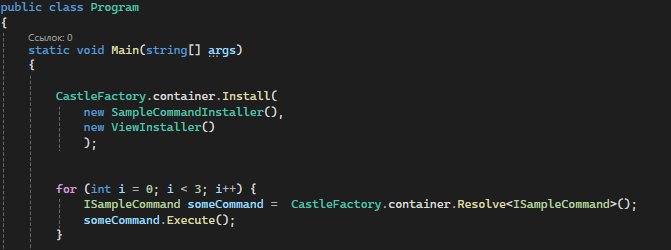




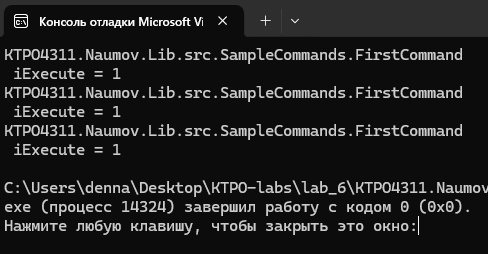
1. Добавляем в класс Program конфигурацию контейнера в начало метода main:



1. Добавляем в класс Program код для вызова команд

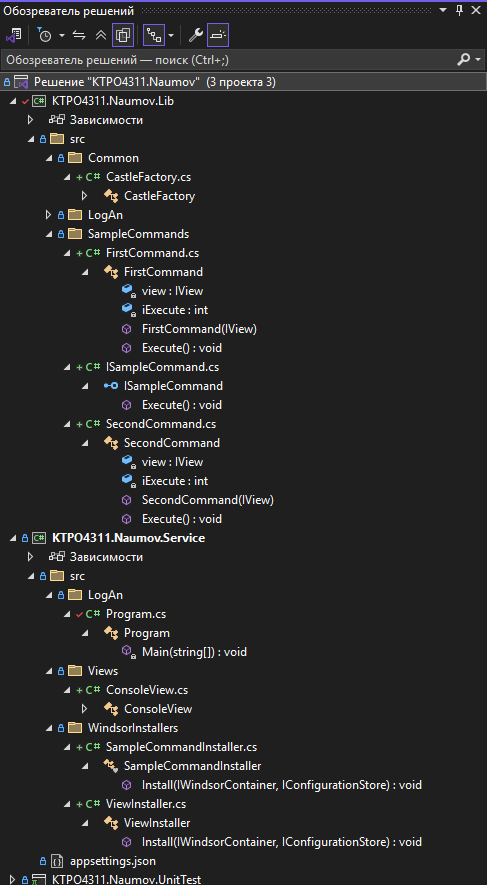


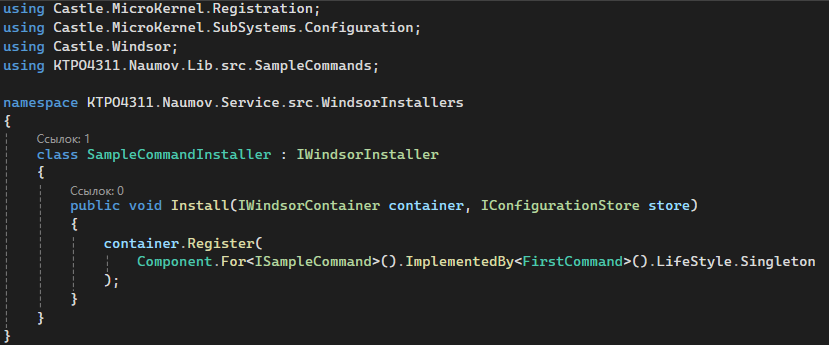
1. Собираем решение и запускаем проект .Service

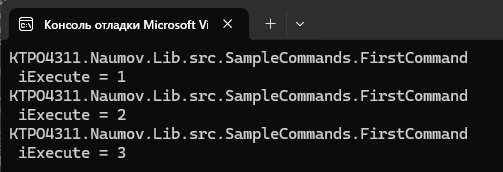


Значение счётчика для всех обращений равно 1, так как в WindsorInstaller используем Transient lifestyle, который создаёт новый экземпляр объекта при каждом вызове Resolve(). Поэтому поле iExecute каждый раз инициализируется значением 0 и после вызова Execute() становится равным 1. Для получения значений 1, 2, 3 нужно использовать Singleton lifestyle, который создаст один экземпляр объекта на все время жизни контейнера.

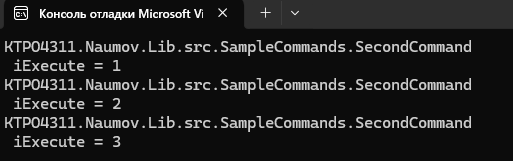
1. Фиксируем состояние окна обозревателя решений



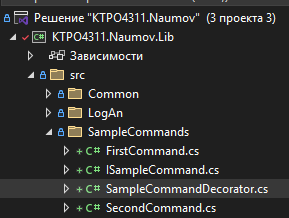
1. Заменим в регистрации зависимость для команды жизненный цикл объекта на Singleton. Он создаёт один экземпляр объекта на все время жизни контейнера.

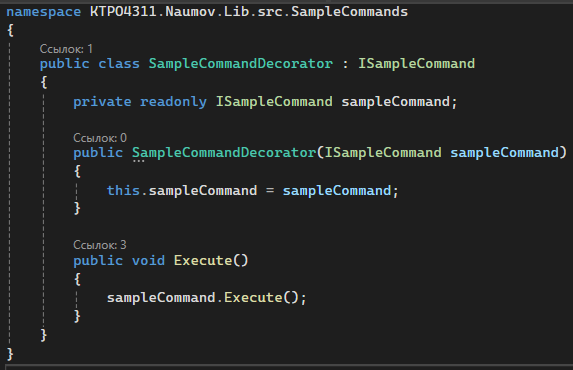


1. Заменим в регистрации зависимость на SecondCommand. Запустим проект.

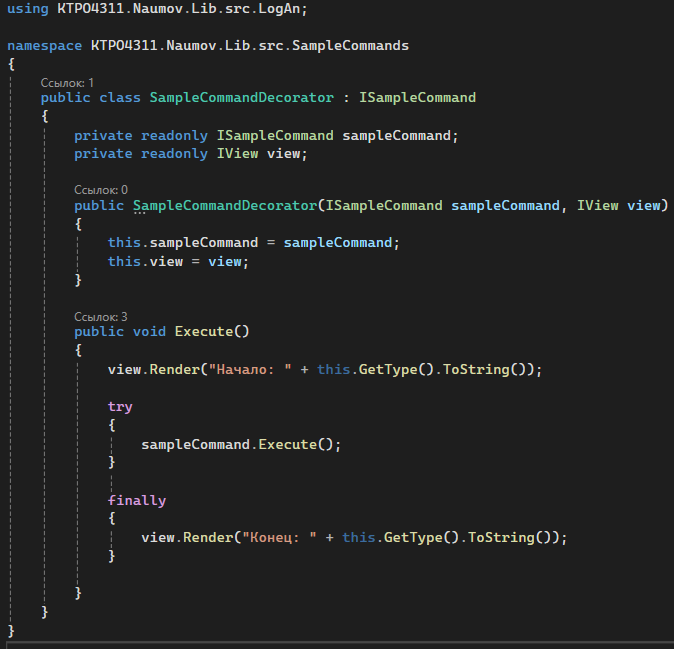


1. **Добавление декоратора.**
2. Создадим в папке SampleCommands проекта .Lib класс SampleCommandDecorator, реализующий интерфейс ISampleCommand.

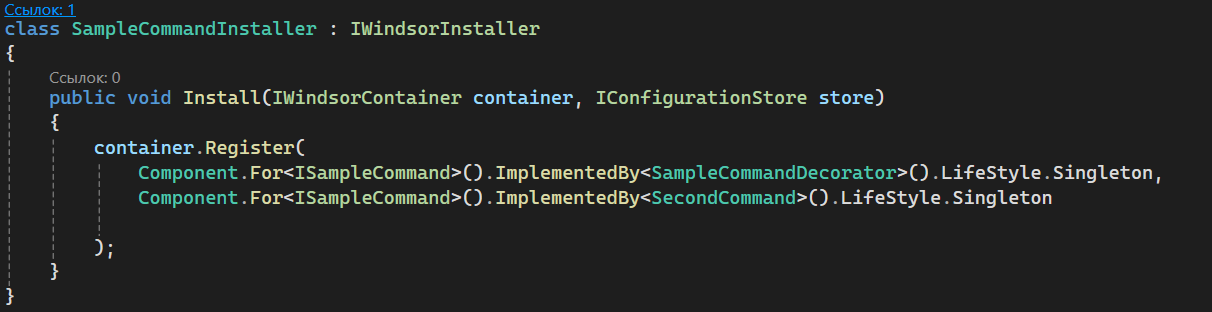




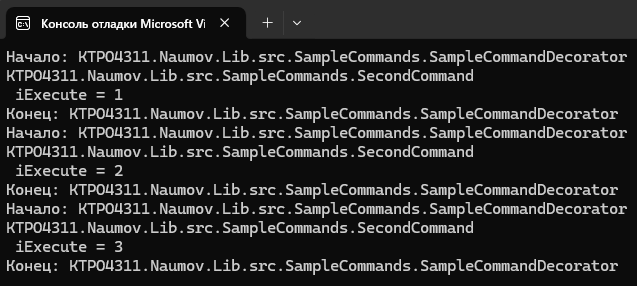
1. Добавляем функциональность декоратора. Через конструктор добавим в него объект IView. Также добавим вывод текста.

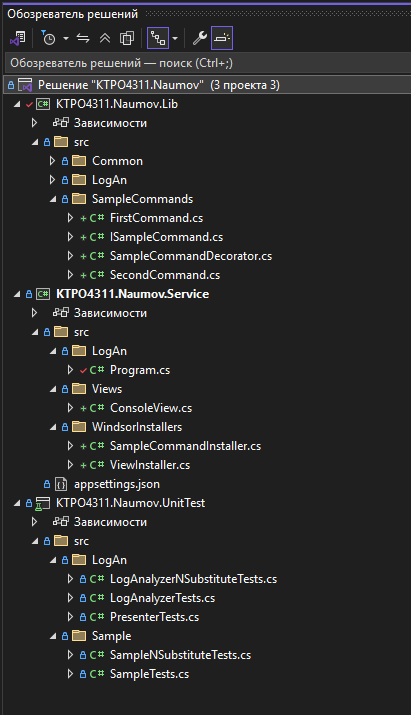


1. Добавляем регистрацию декоратора

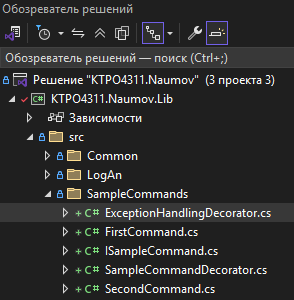


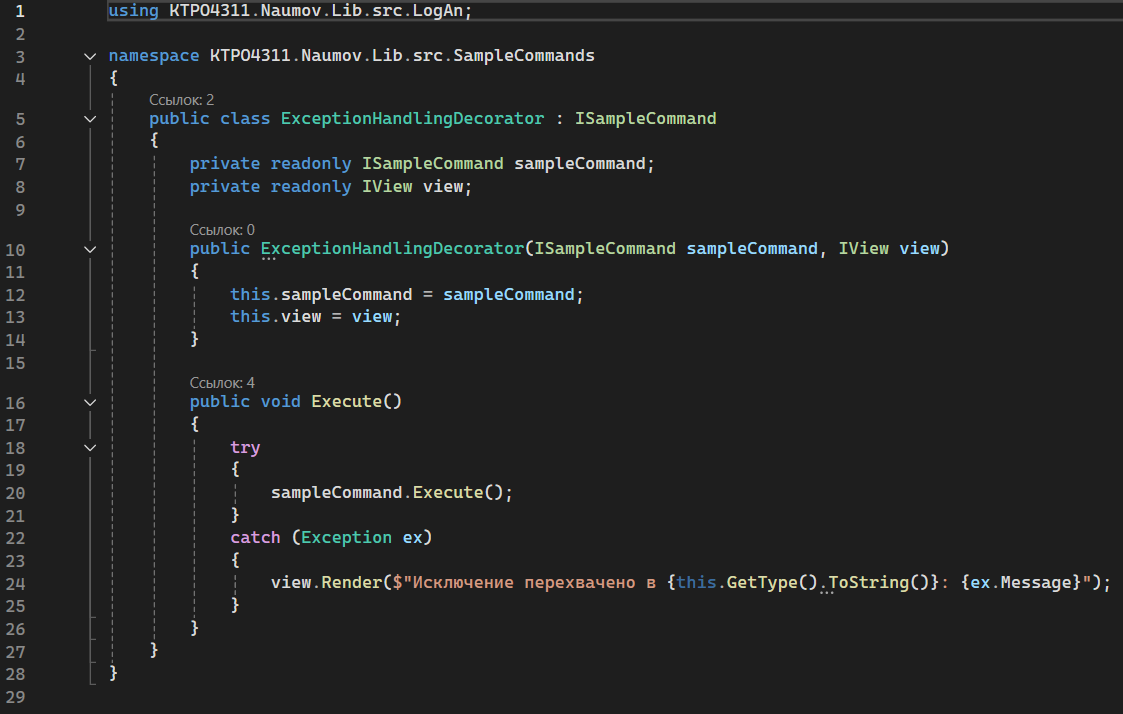
1. Запустим проект .Service

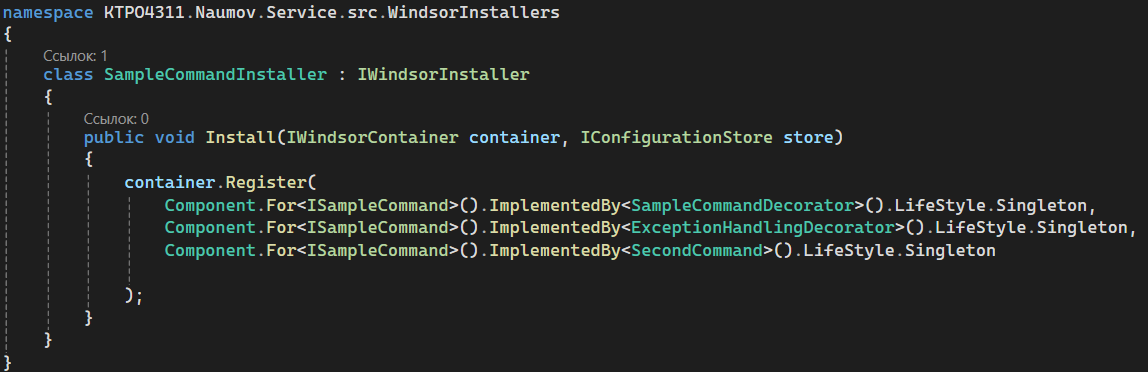




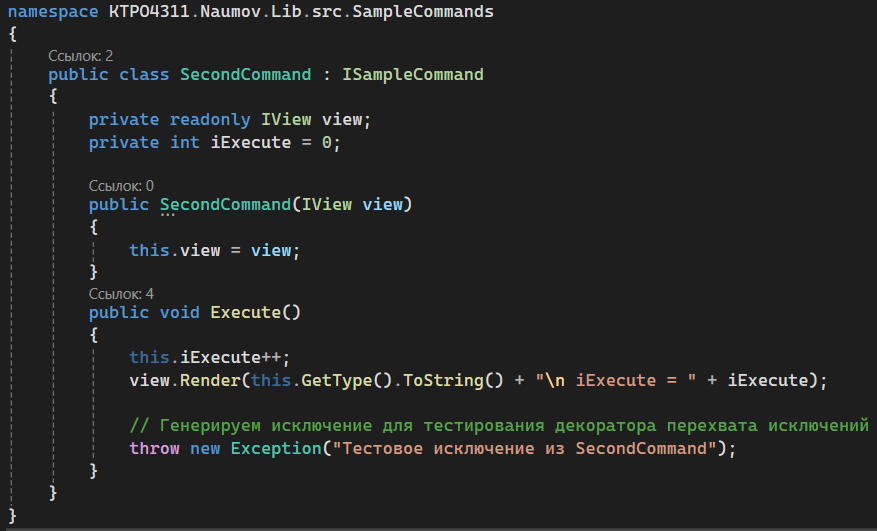
1. **Добавление декоратора для перехвата исключений.**
2. В проекте .Lib в папке src/SampleCommands создадим класс ExceptionHandlingDecorator. Регистрируем его.



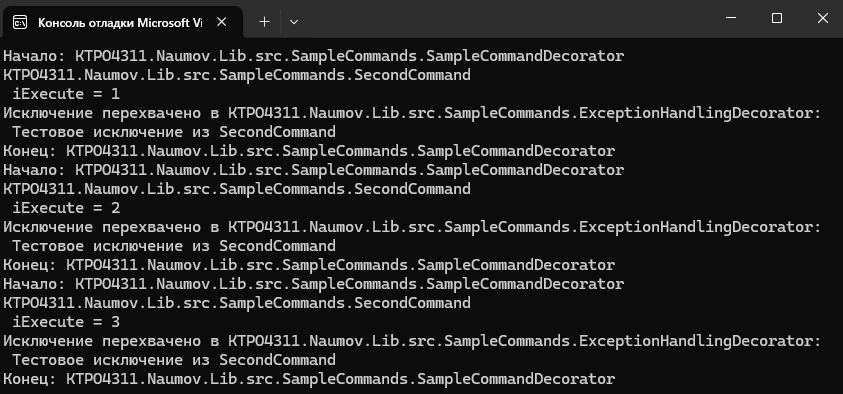




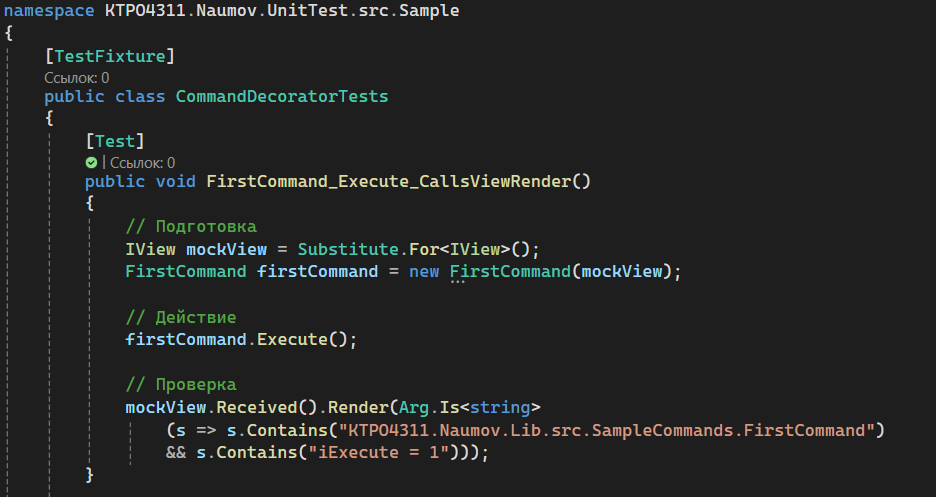
1. Добавим в декорируемую команду SecondCommand вызов исключения



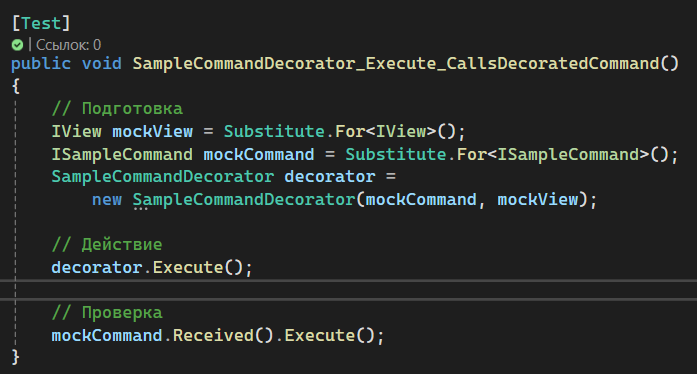
1. Запустим проект .Service.



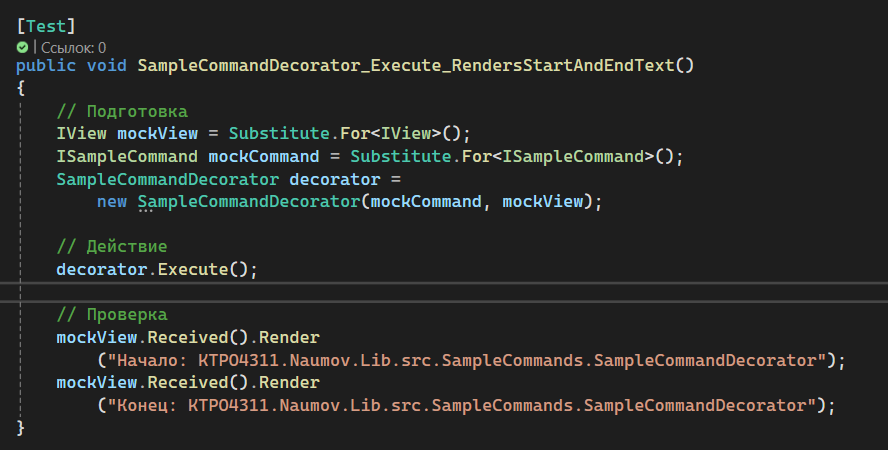
1. **Реализация автономных тестов для разработанных классов.**
2. Создадим в папке src/Sample проекта .UnitTest класс CommandDecoratorTests
3. Создадим тест для команды FirstCommand: метод Execute() вызывает вывод текста согласно заданию



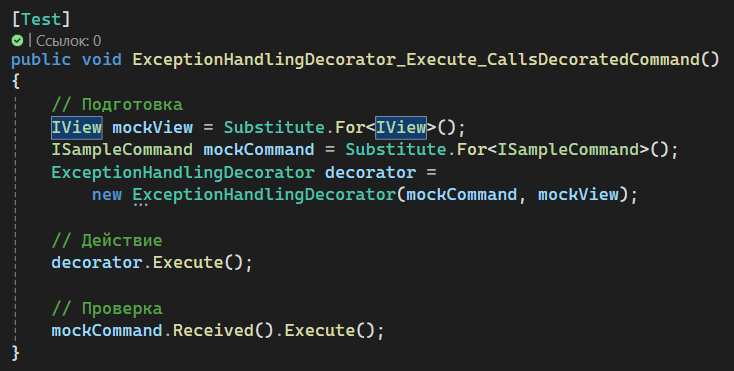
1. Создадим тест для декоратора SampleCommandDecorator: метод Execute() вызывает метод декорируемого объекта.



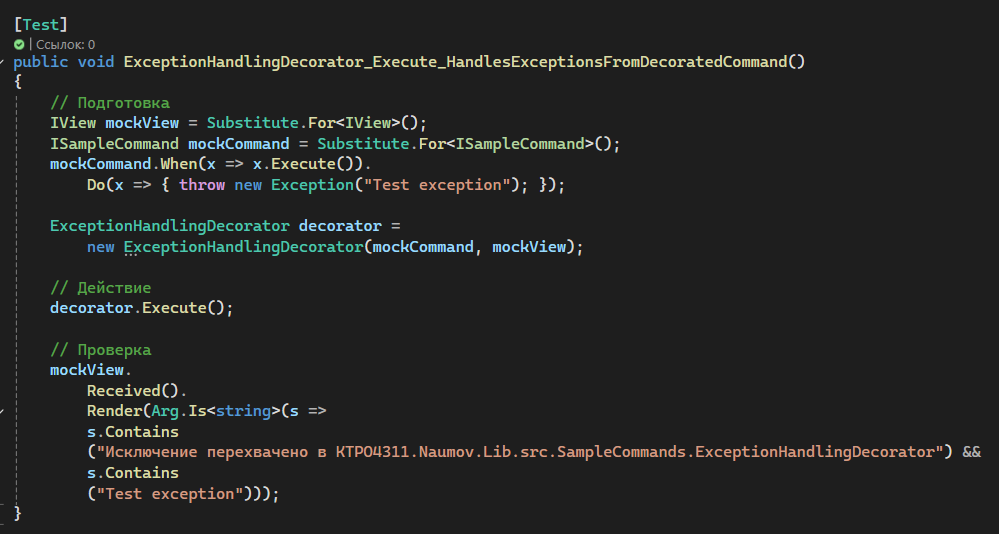
1. Создадим тест для декоратора SampleCommandDecorator: метод Execute() вызывает вывод текста согласно заданию.



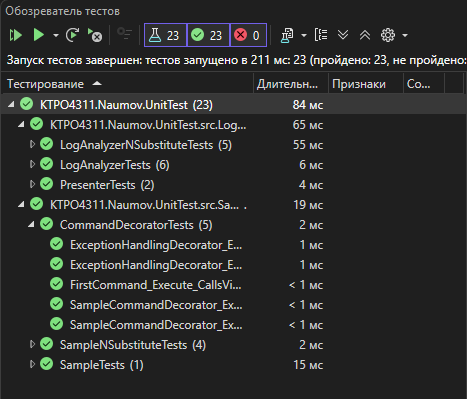
1. Создадим тест для декоратора ExceptionCommandDecorator: метод Execute() вызывает метод декорируемого объекта.

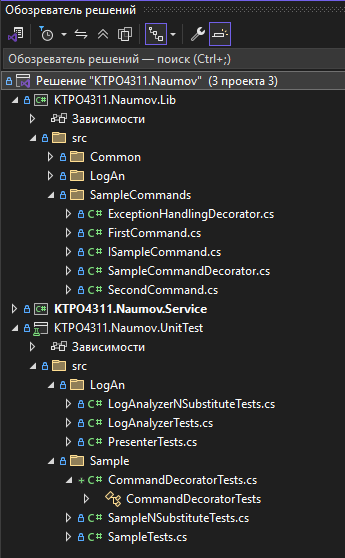


1. Создадим тест для декоратора SampleCommandDecorator: метод Execute() обрабатывает исключения, возникшие в декорируемом объекте



1. Выполним тесты и зафиксируем состояние окна обозревателя решений.





**Выводы**

В ходе лабораторной работы №5 были приобретены практические навыки использования делегатов и событий, а также тестирования операций, связанных с событиями, с применением тестового каркаса NUnit и изолирующего каркаса NSubstitute. Цель работы заключалась в изучении и реализации механизма генерации событий в классе LogAnalyzer, подписки на события через класс Presenter, а также создании тестов для проверки корректности обработки событий. Были реализованы сценарии, включая ручную имитацию событий с использованием наследования и динамическую имитацию с помощью NSubstitute, что позволило обеспечить изолированное тестирование модулей.