**Лабораторная работа №6.**

**Тестирование событий**

**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

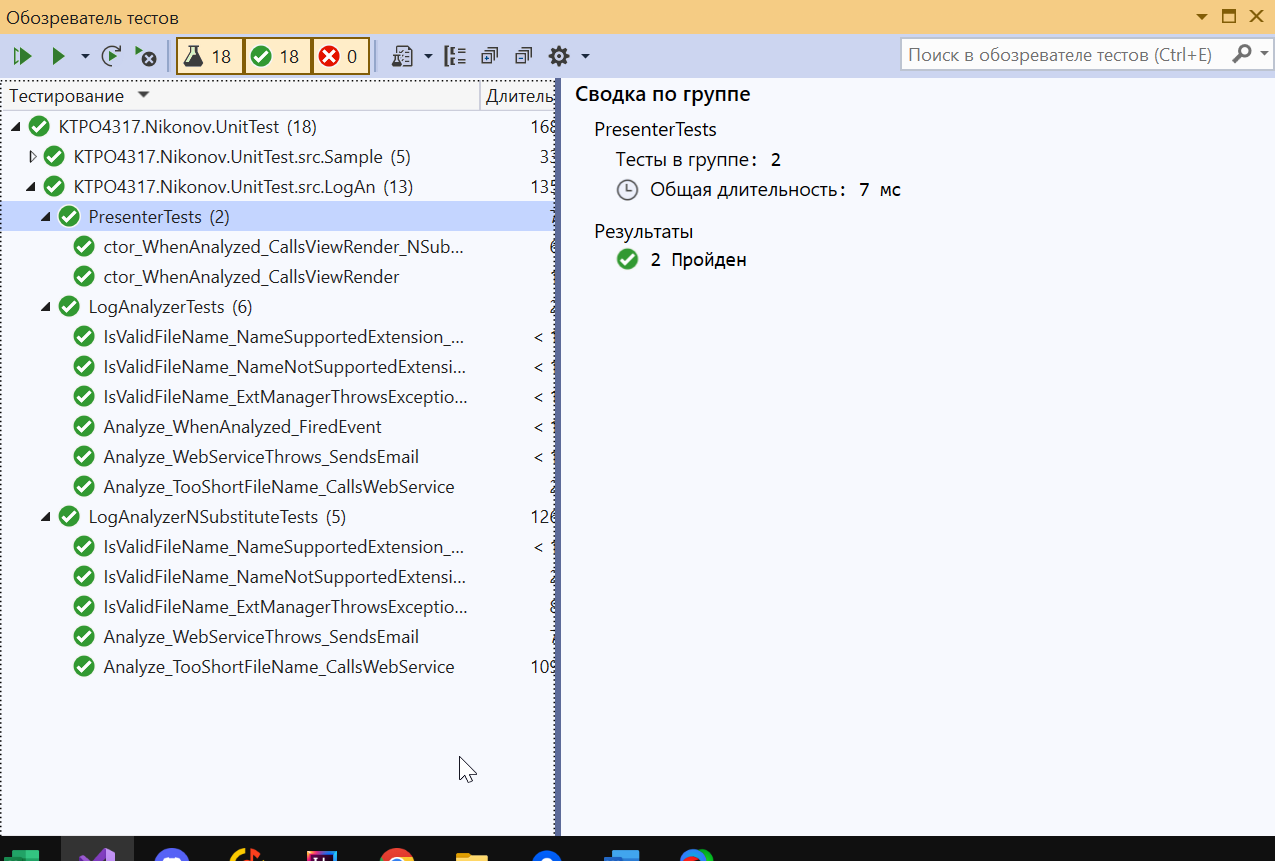
8. На каждом шаге делайте снимки исходного кода создаваемых или изменяемых классов и тестов, окна «Результаты тестов» и «Обозреватель решения» и сохраните в документе MS Word.

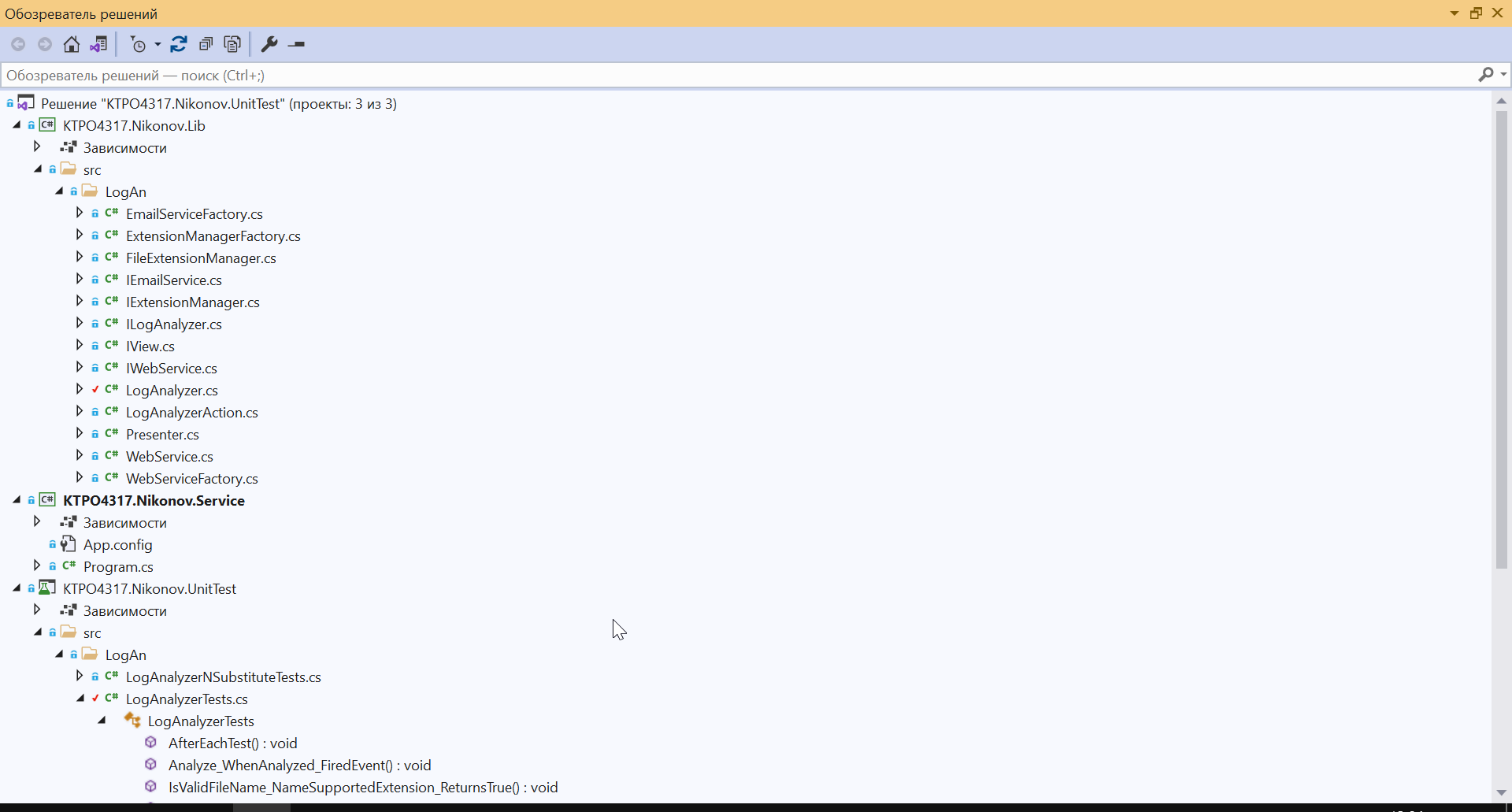
9. Оформить отчет

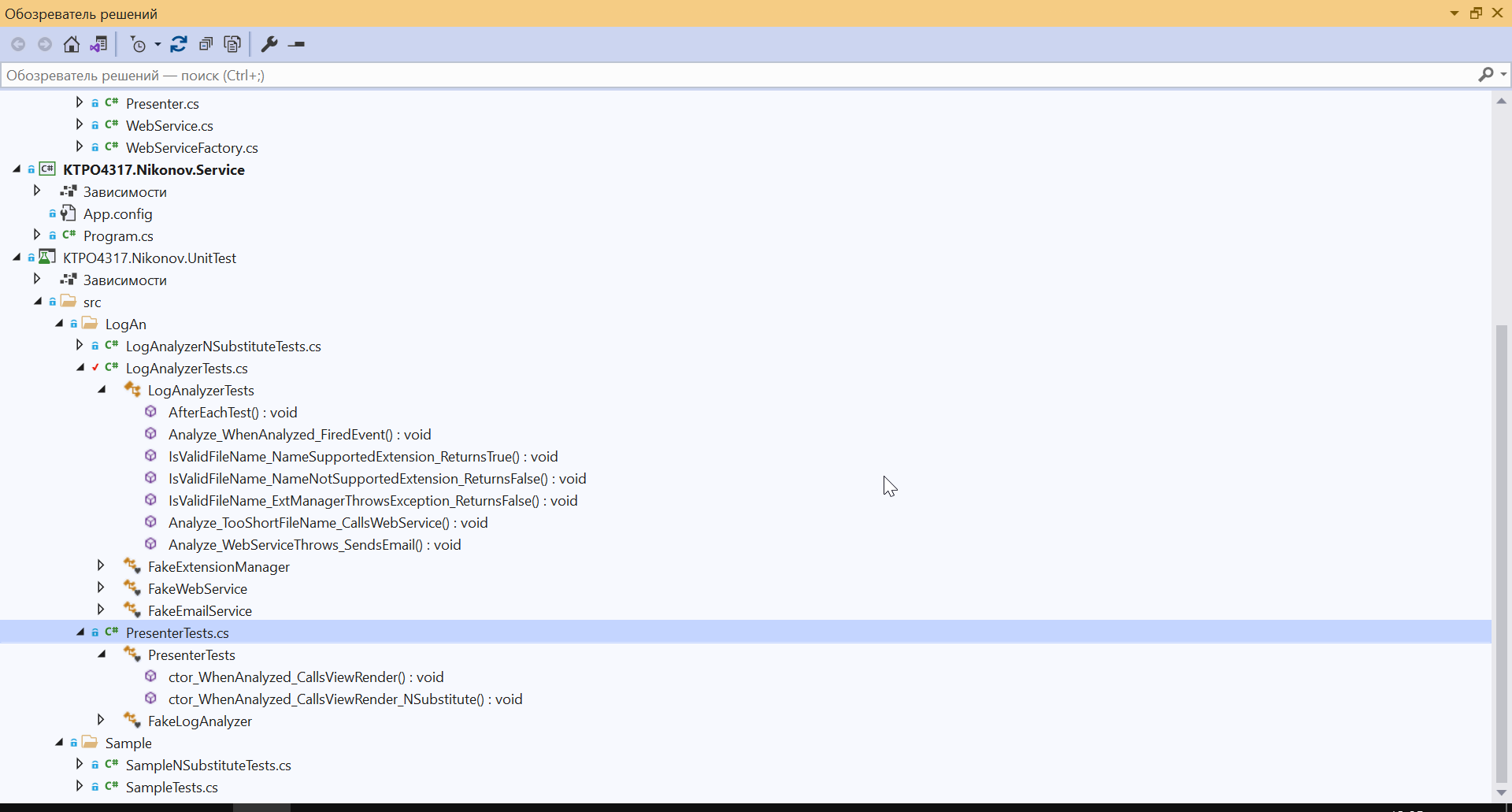
**Ход работы**

**1. Подготовка проекта**

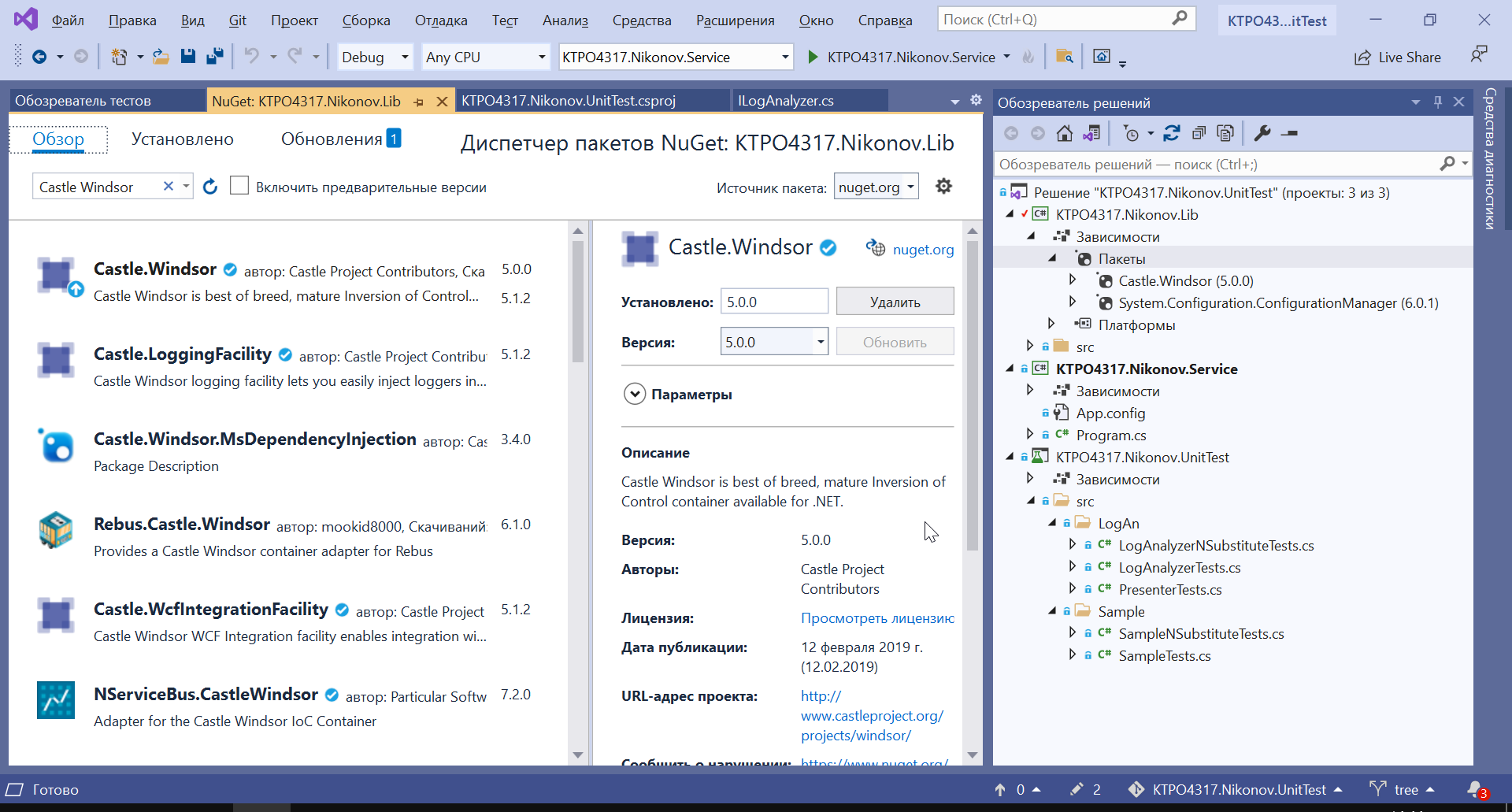
Для выполнения данной лабораторной работы возьмите решение, полученное в результате выполнения лабораторной работы №5. Выполните тесты. Зафиксируйте исходное состояние окна «Обозреватель решения», окна «Результаты тестов»: и сохраните в документе MS Word.







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

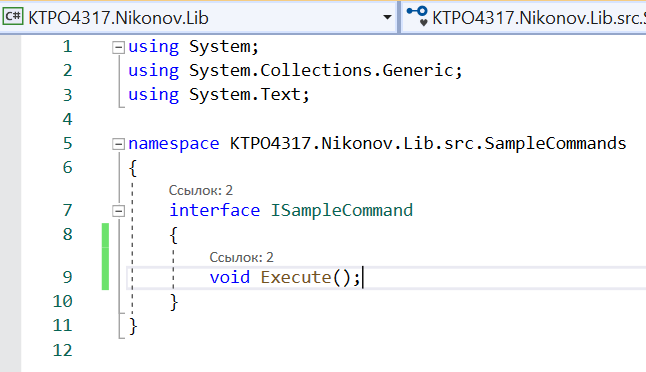


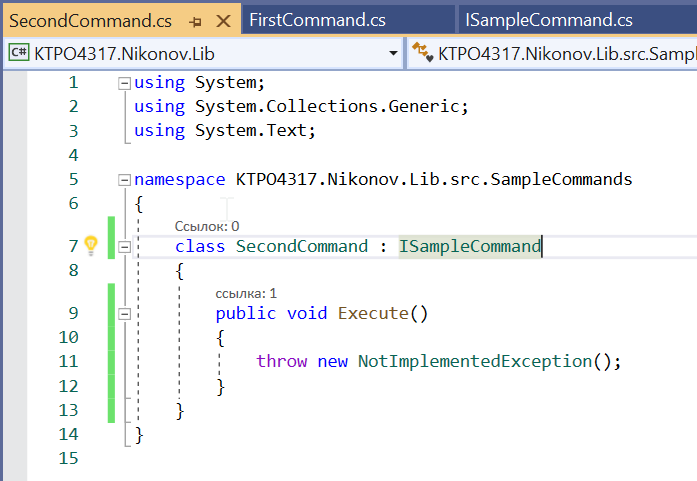
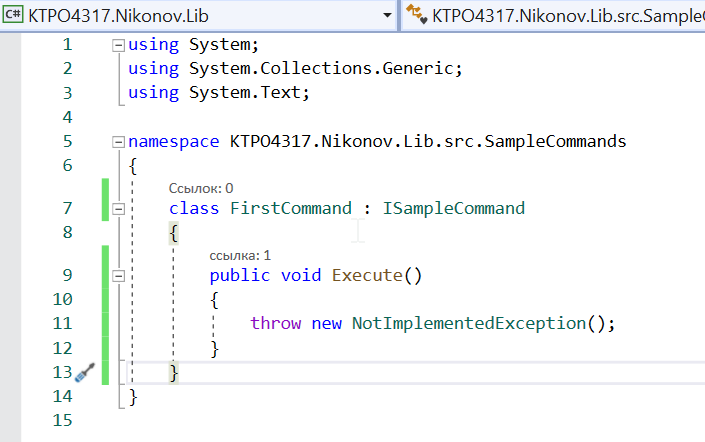
**3. Реализация паттерна команда**

В проекте «.Lib» создайте папку «src/SampleCommands»

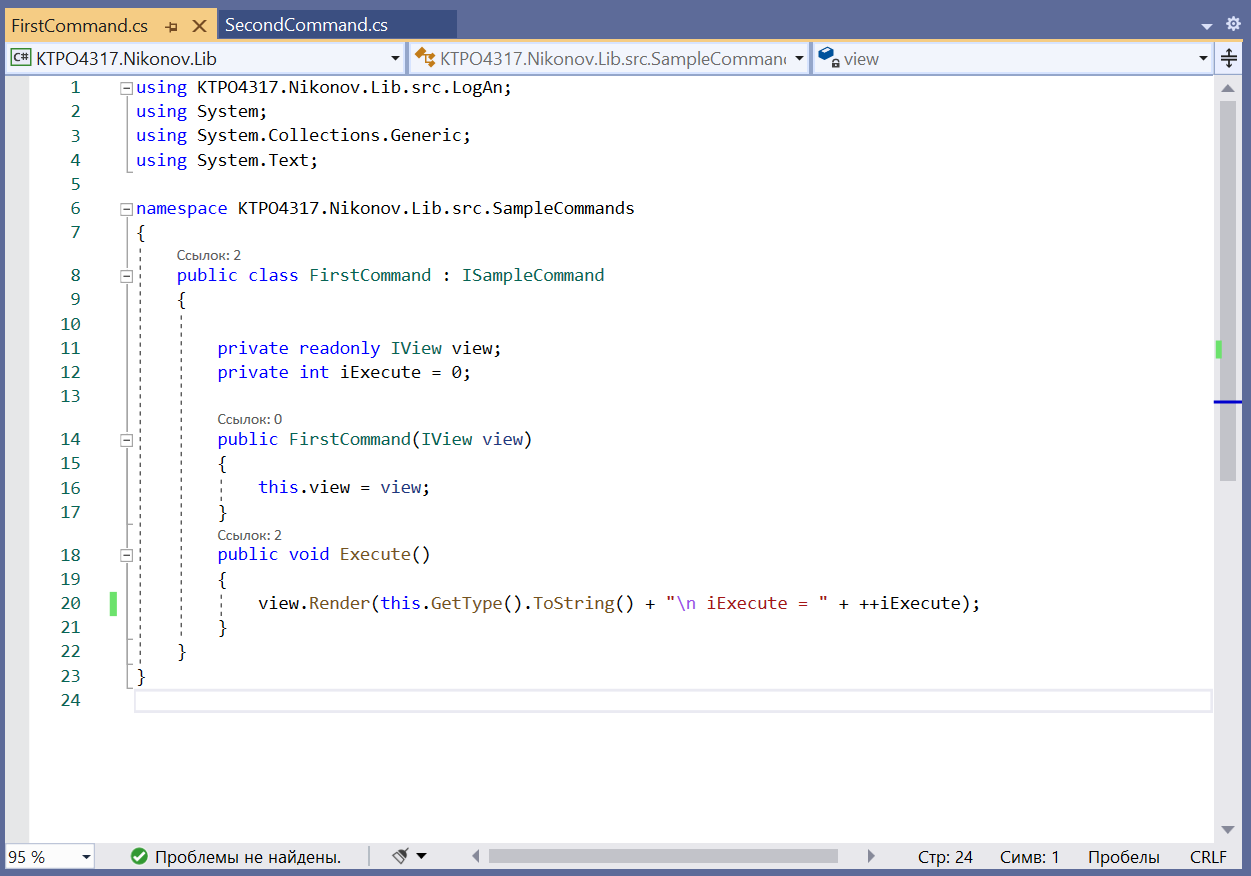
Создайте в ней интерфейс соответствующий шаблону команда:

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

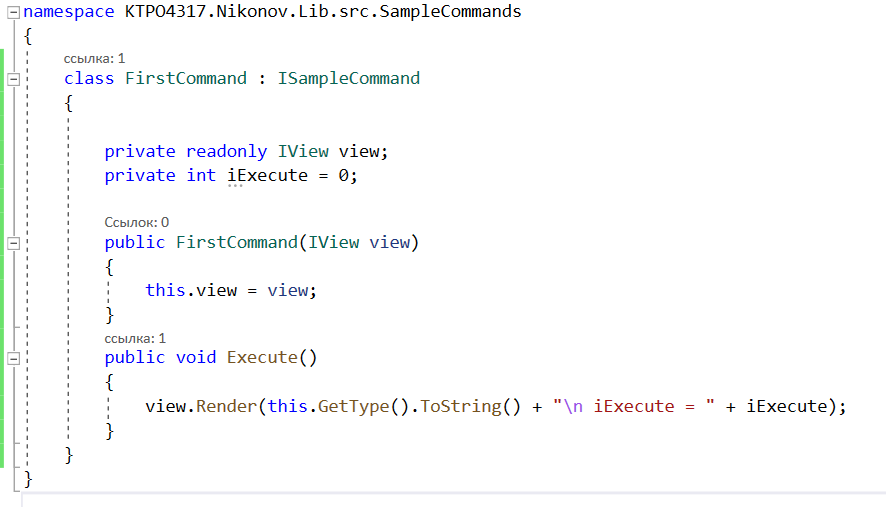




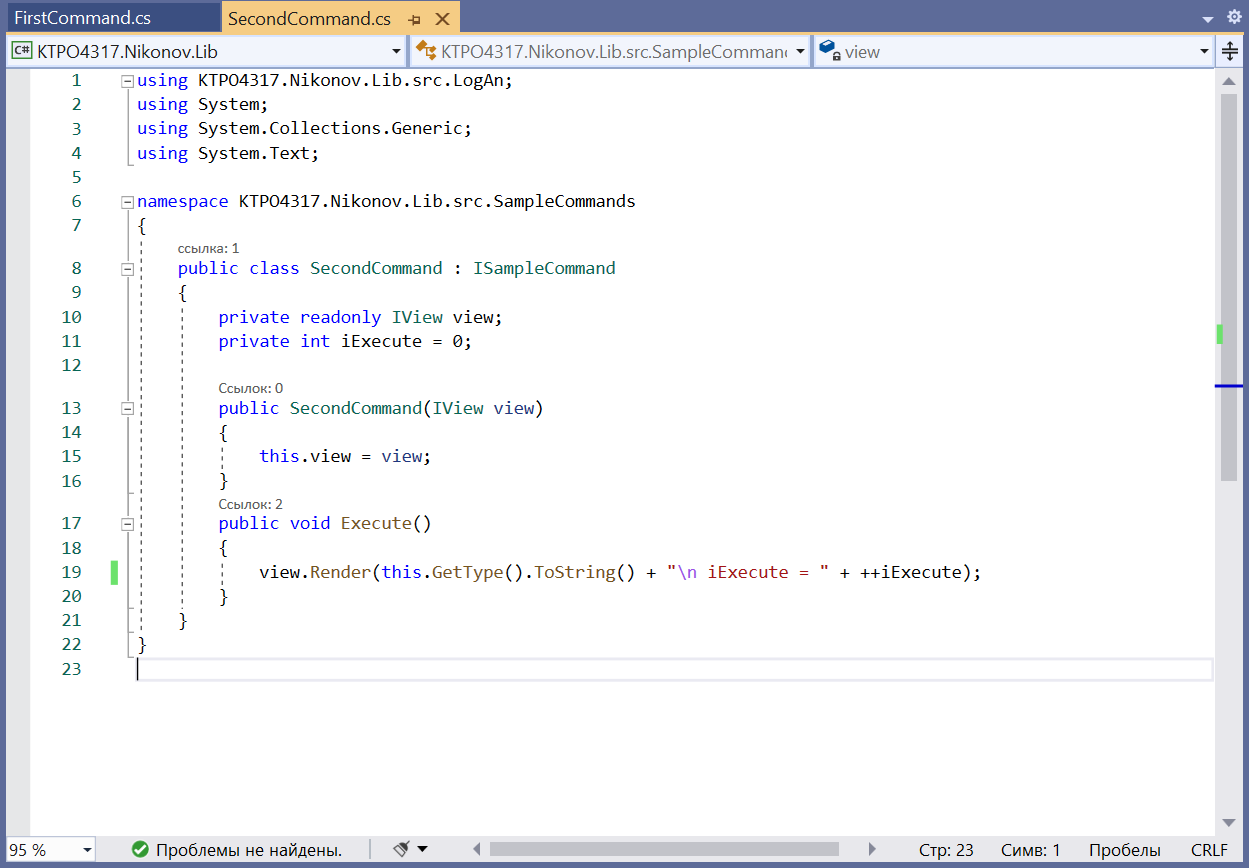
Пусть команда должна уметь выводить имя своего класса, и количество раз вызова метода Execute() для экземпляра класса. Вывод будем осуществлять с помощью интерфейса IView, созданного в предыдущей лабораторной работе. Будем использовать внедрение зависимости через конструктор. Для подсчета количества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



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



Аналогичным образом реализуйте SecondCommand.

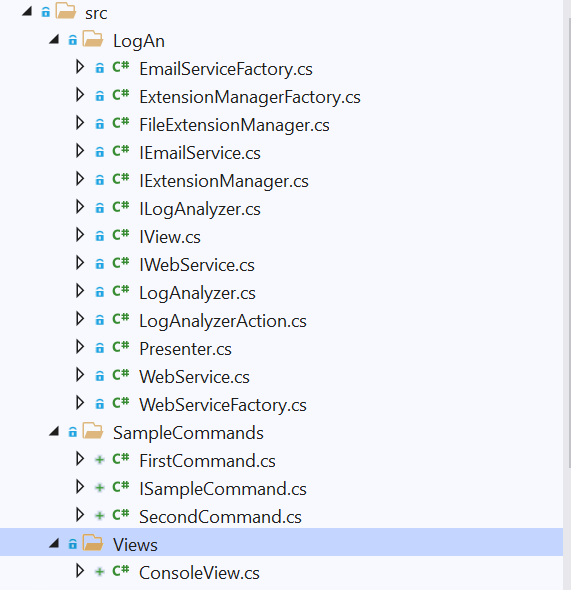


В проекте «.Service» создайте папку «src\Views»

В этой папке создайте класс ConsoleView, реализующий интерфейс View.

Зафиксируйте в отчете код классов и состояние окна Обозреватель решения.

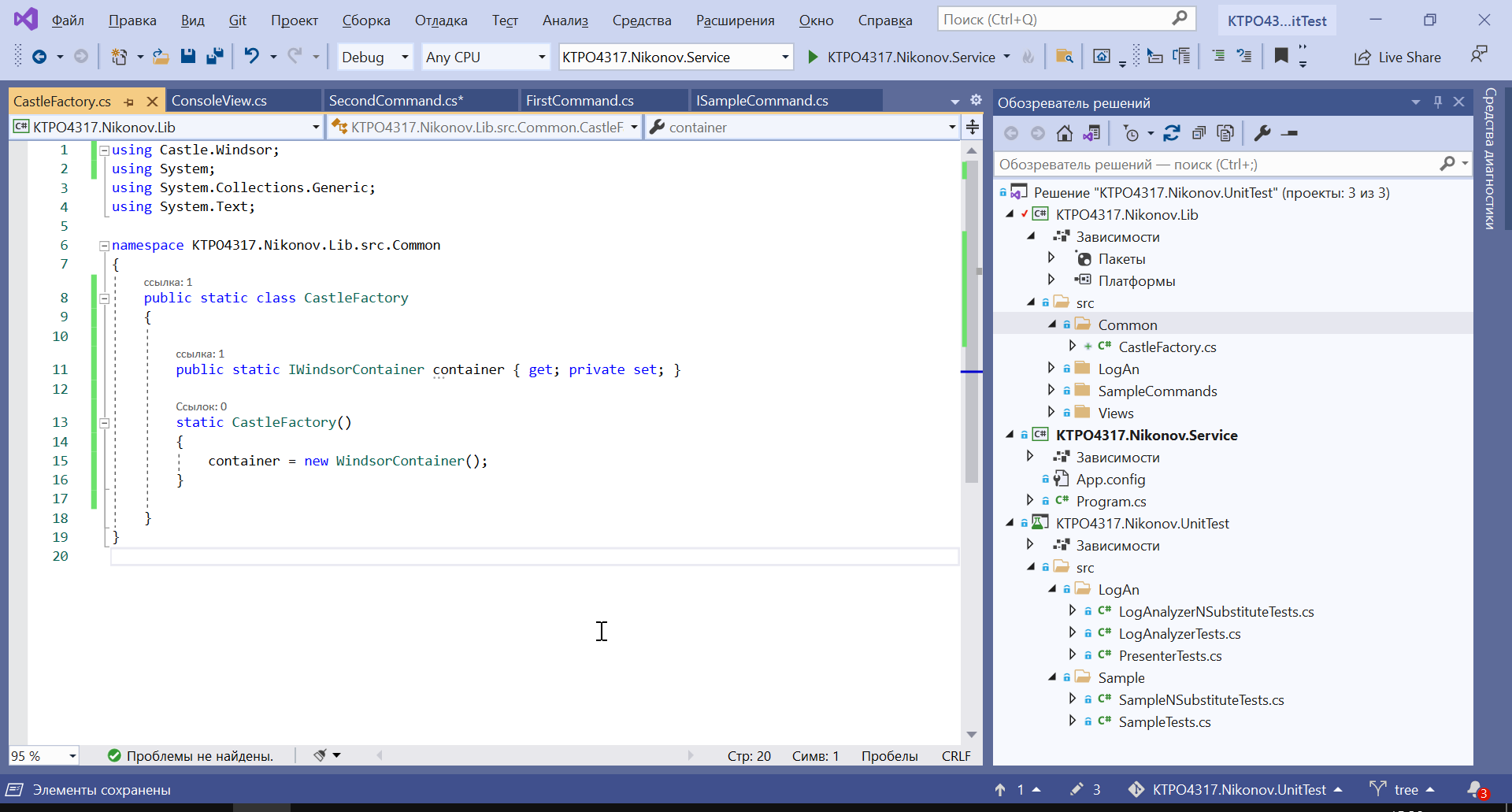




**4. Добавление и конфигурация контейнера**

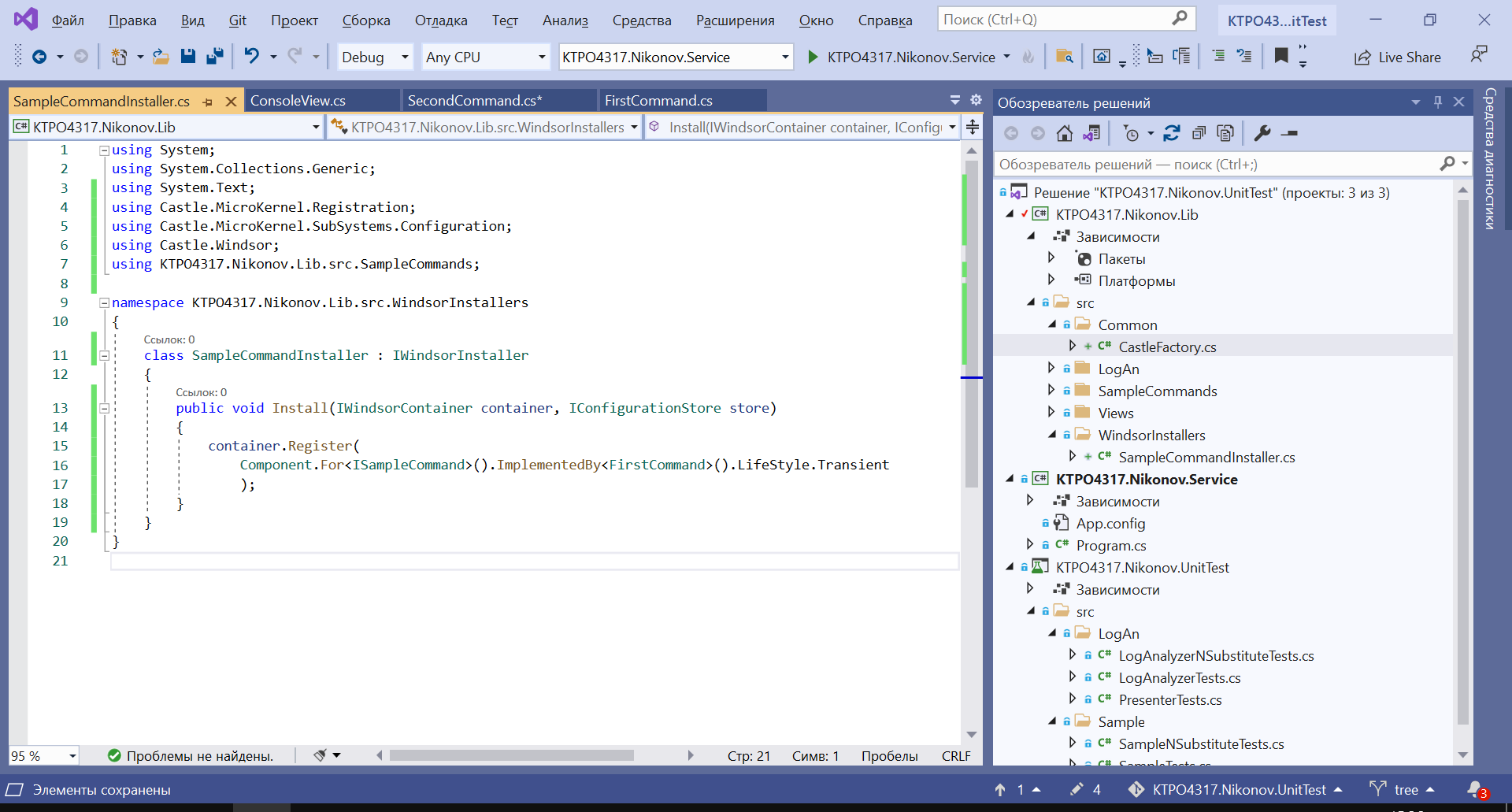
В проекте «.Lib» создайте папку «src/Common».

Создайте в ней статический класс CastleFactory. В нем разметим ссылку на IoC контейнер.

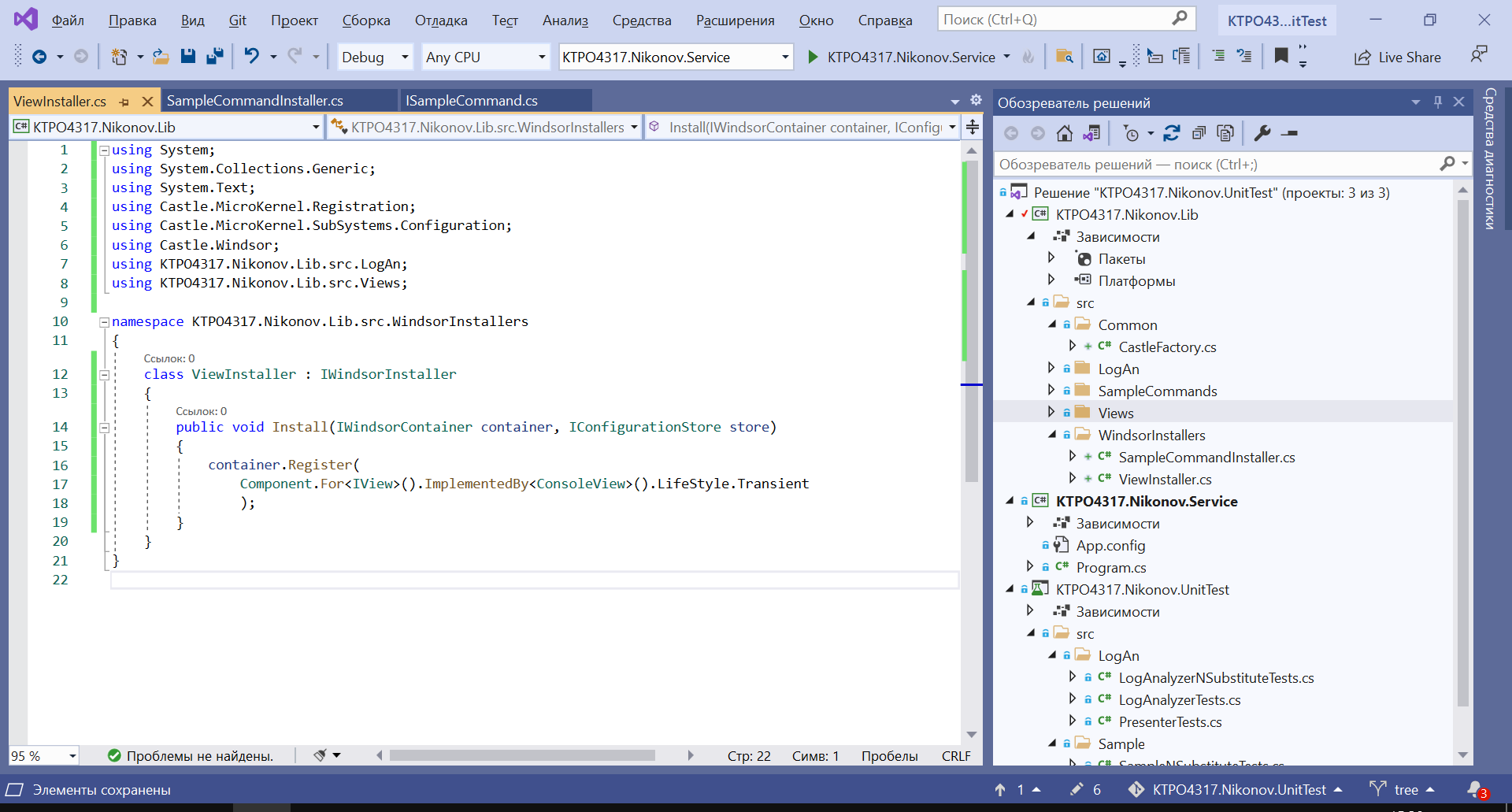


В проекте «.Service» создайте папку «src/WindsorInstallers».

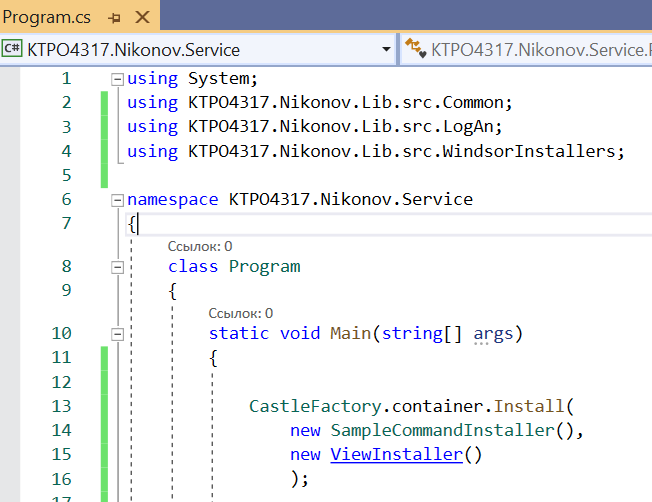
Добавьте в нее класс для конфигурирования «команд».



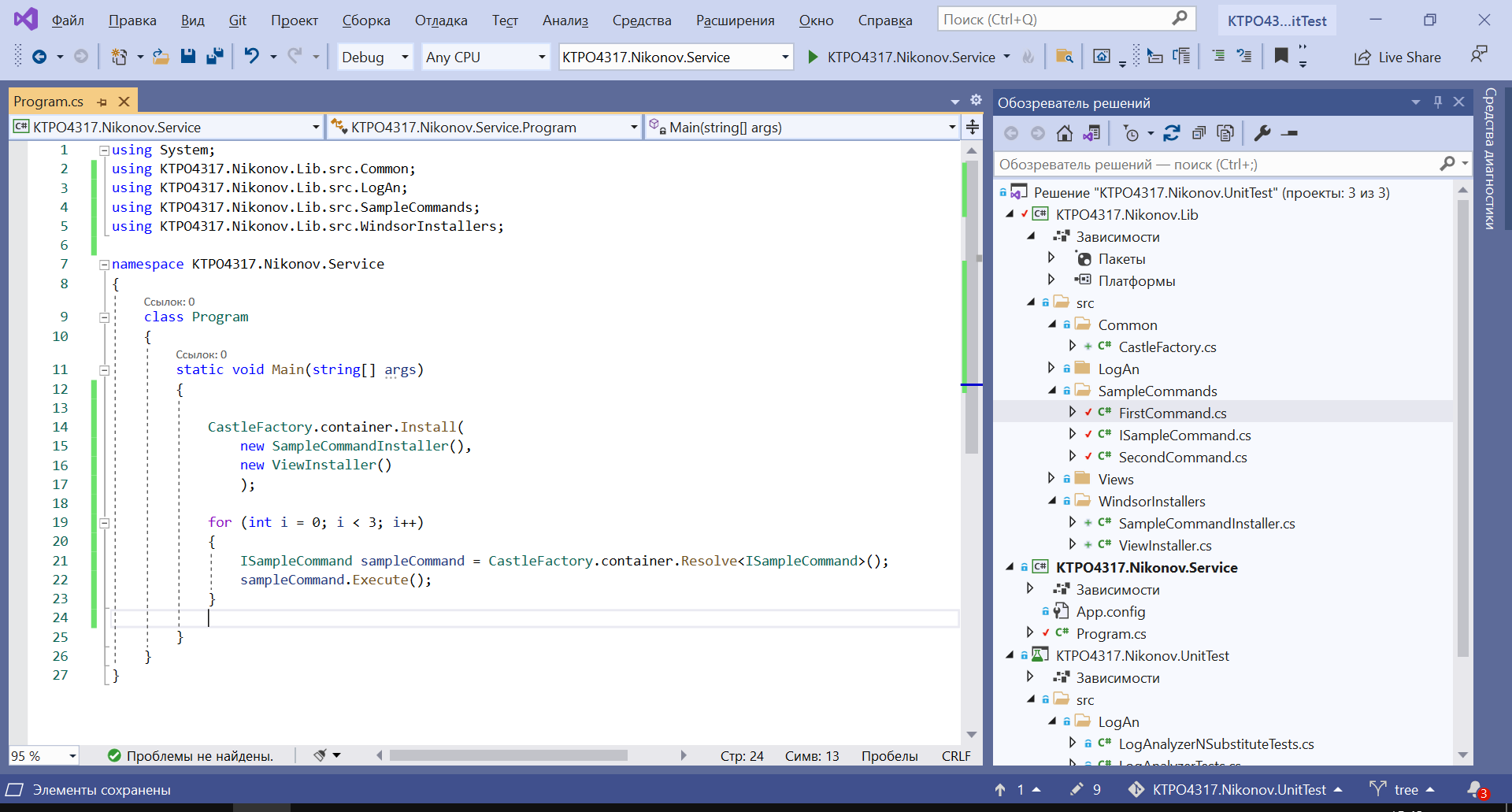
Добавьте в папку «src/WindsorInstallers» класс для конфигурирования «представления» ViewInstaller.

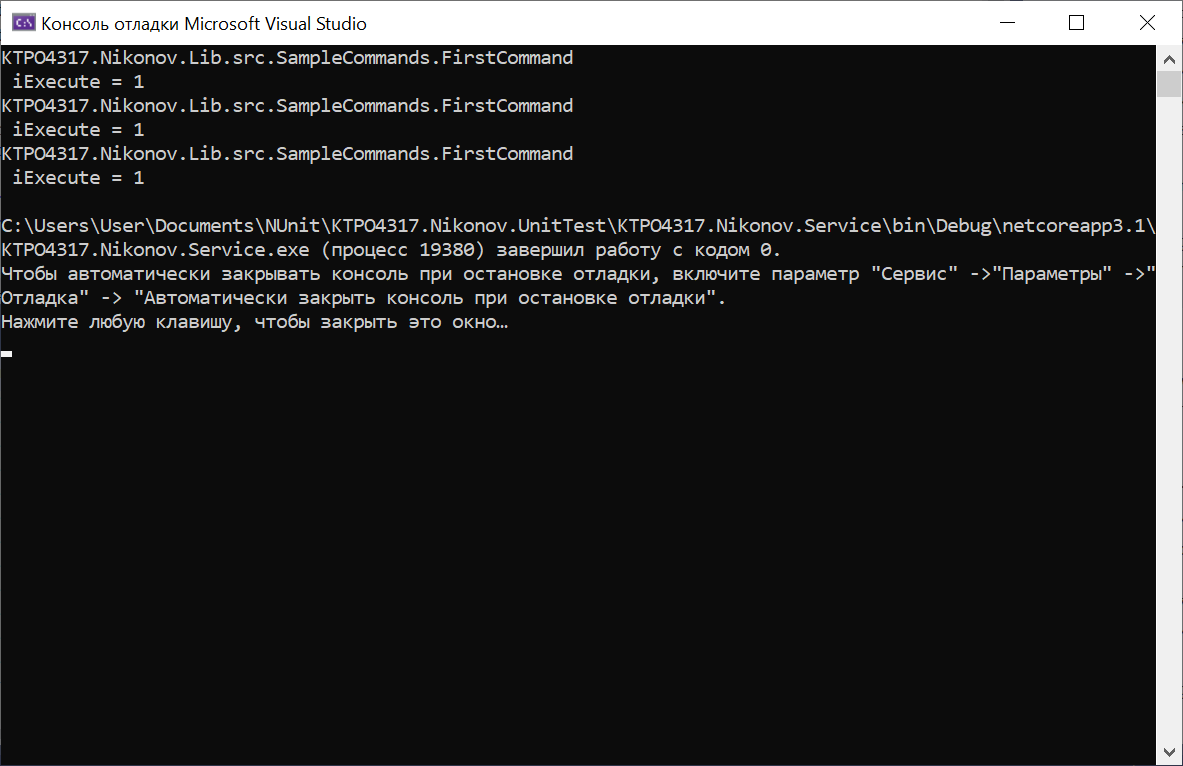


Добавьте в класс «Program» конфигурацию контейнера в начало метода main.

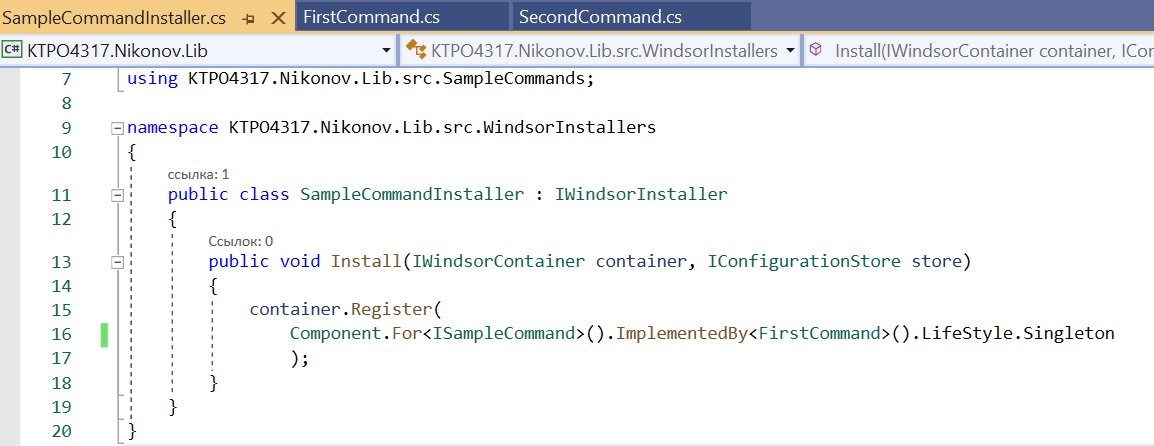


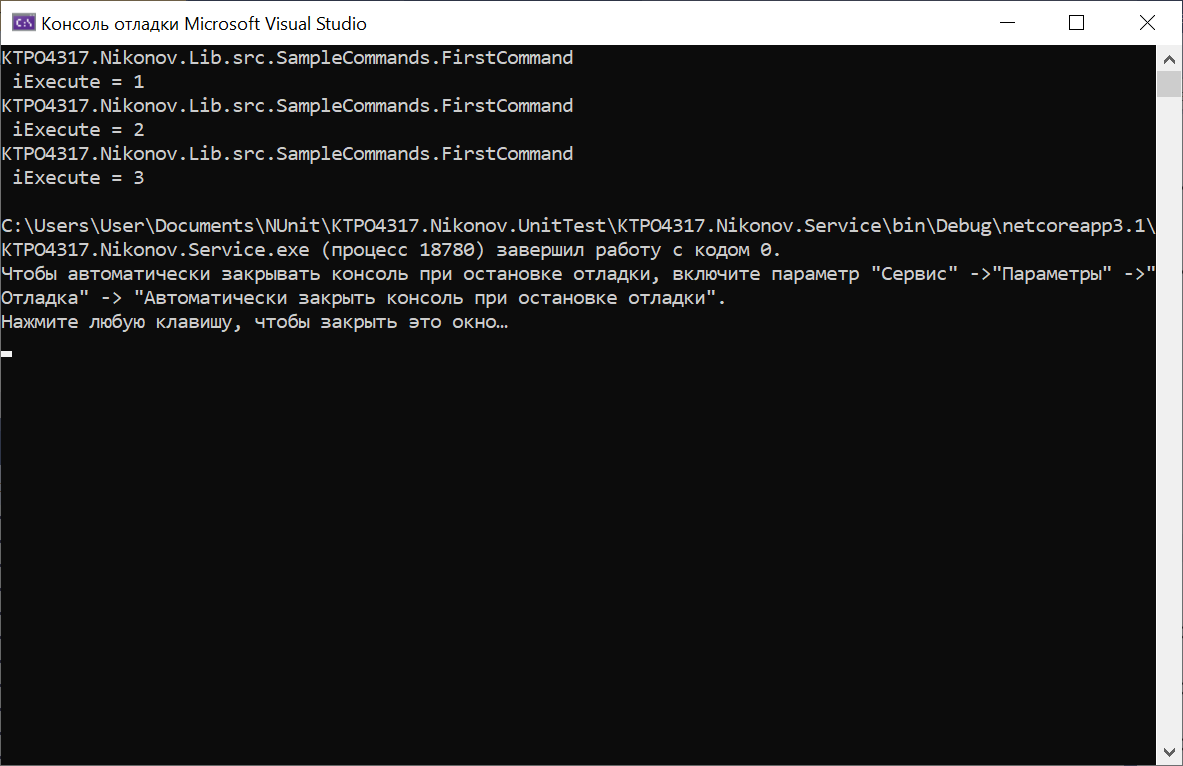
Добавьте в класс «Program» код вызова «команд».



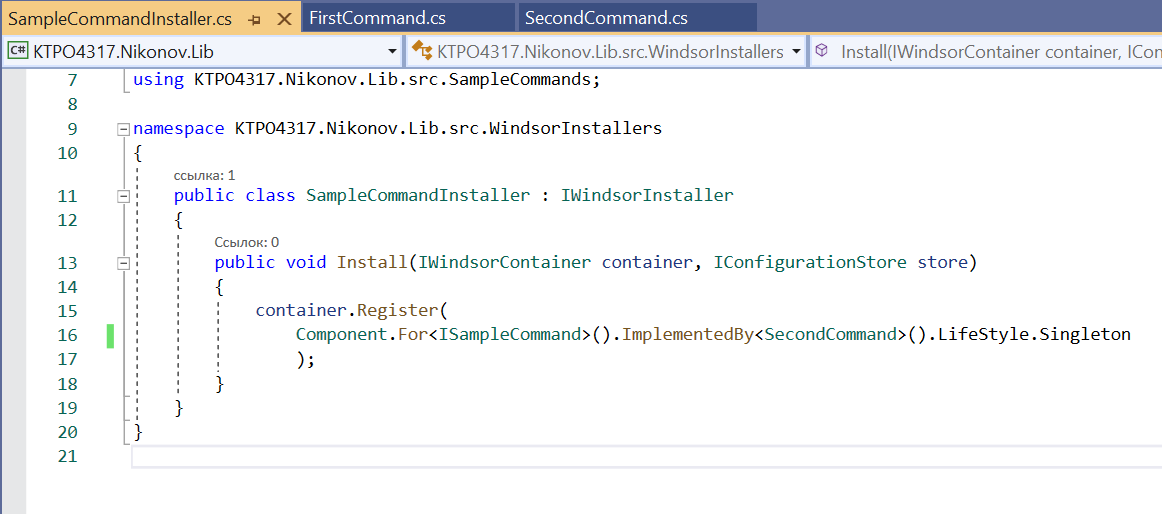


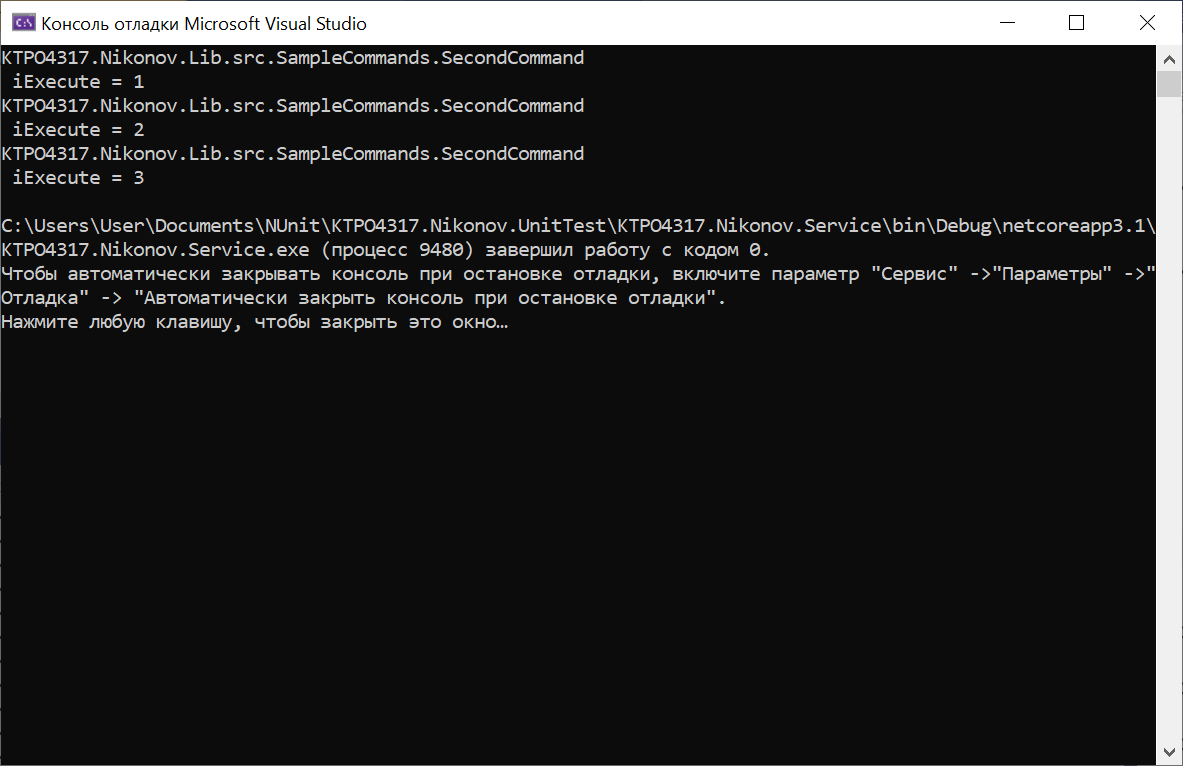
Заменить в регистрации зависимости для команды жизненный цикл объекта на Singleton. Запустите проект «.Service». Значение счетчика изменяется.





Заменить в регистрации зависимости для класс FirstCommand на SecondCommand. Запустите проект «.Service». Зафиксируйте в отчете код измененный классов и результат запуска.



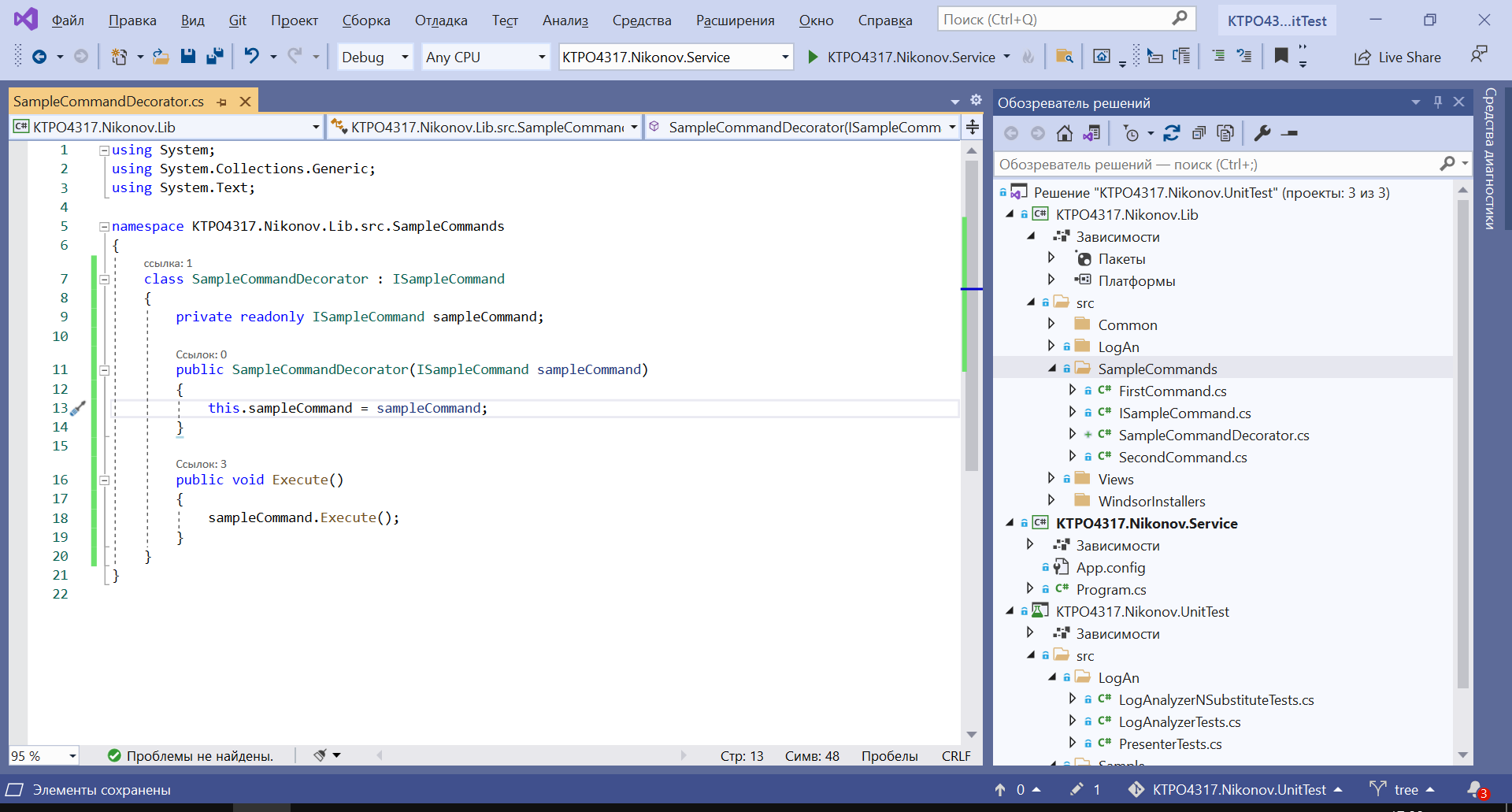


**5. Добавление декоратора**

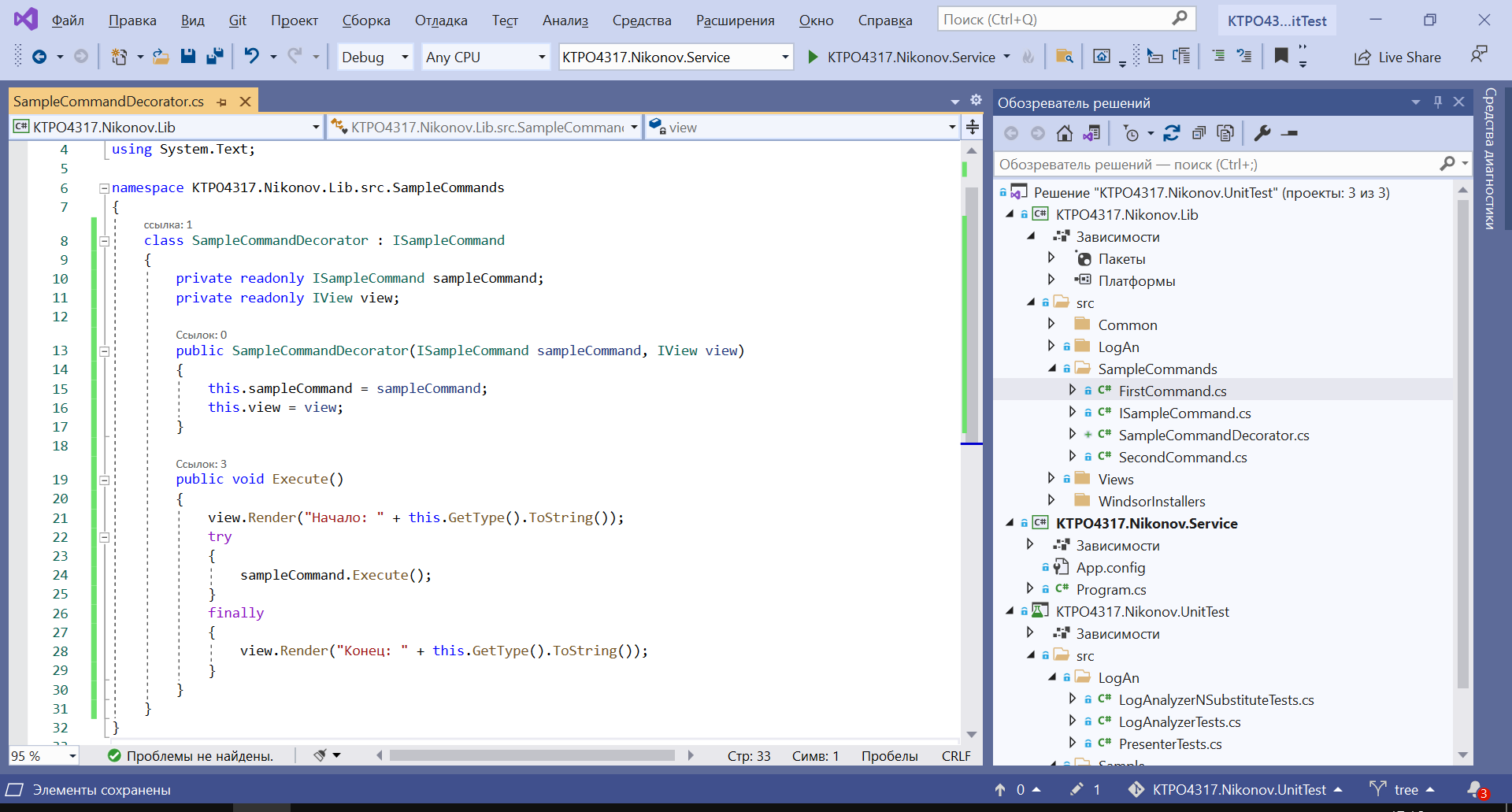
Создайте в папке «SampleCommands» проект «.Lib» класс SampleCommandDecorator реализиующий интерфейс ISampleCommand.

В конструкторе класса в качестве параметра передайте объект класса ISampleCommand, определите и проинициализируйте поле для него.

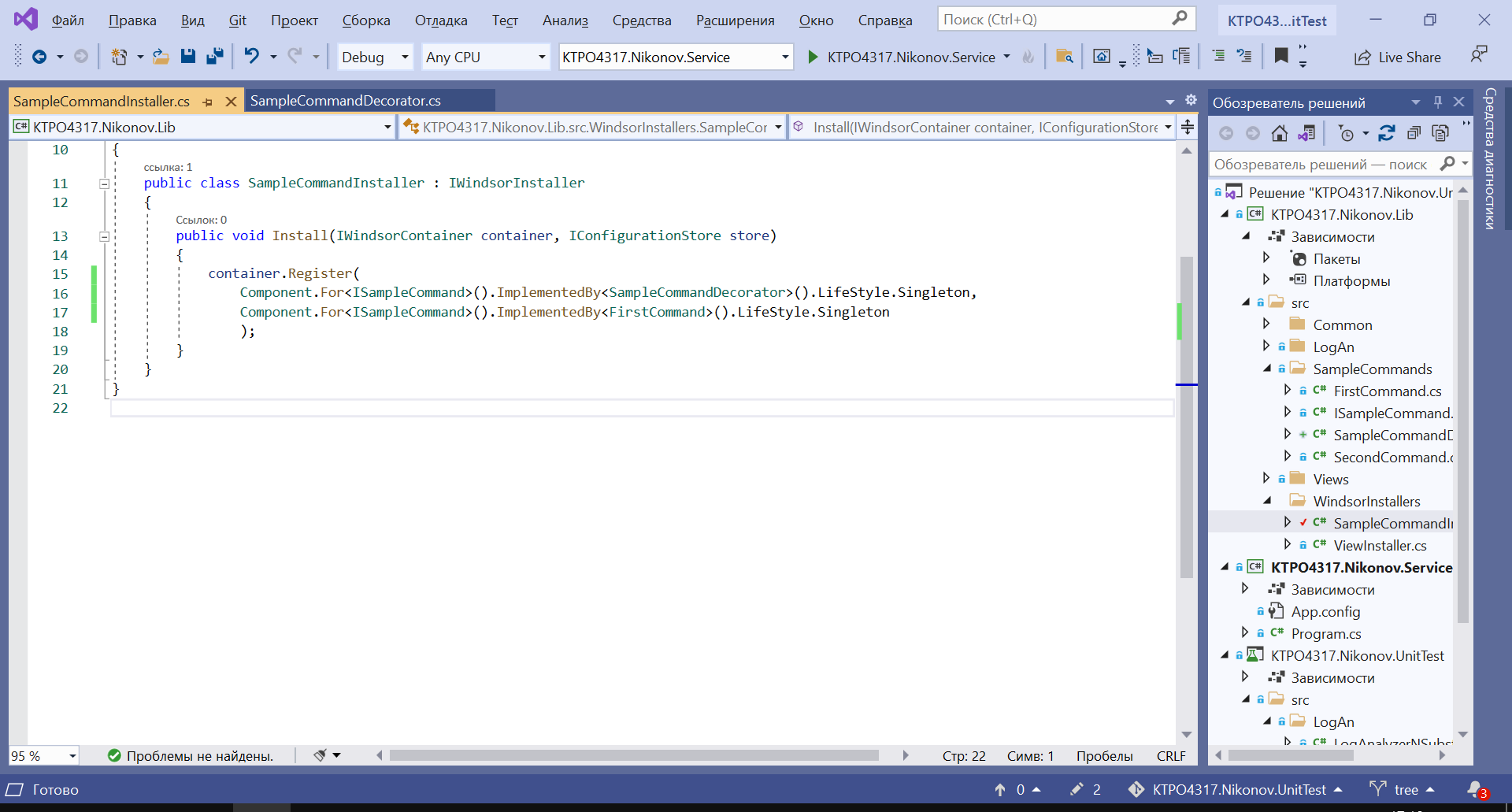
Добавьте в метод Execute() вызов метода объекта переданного в конструкторе.

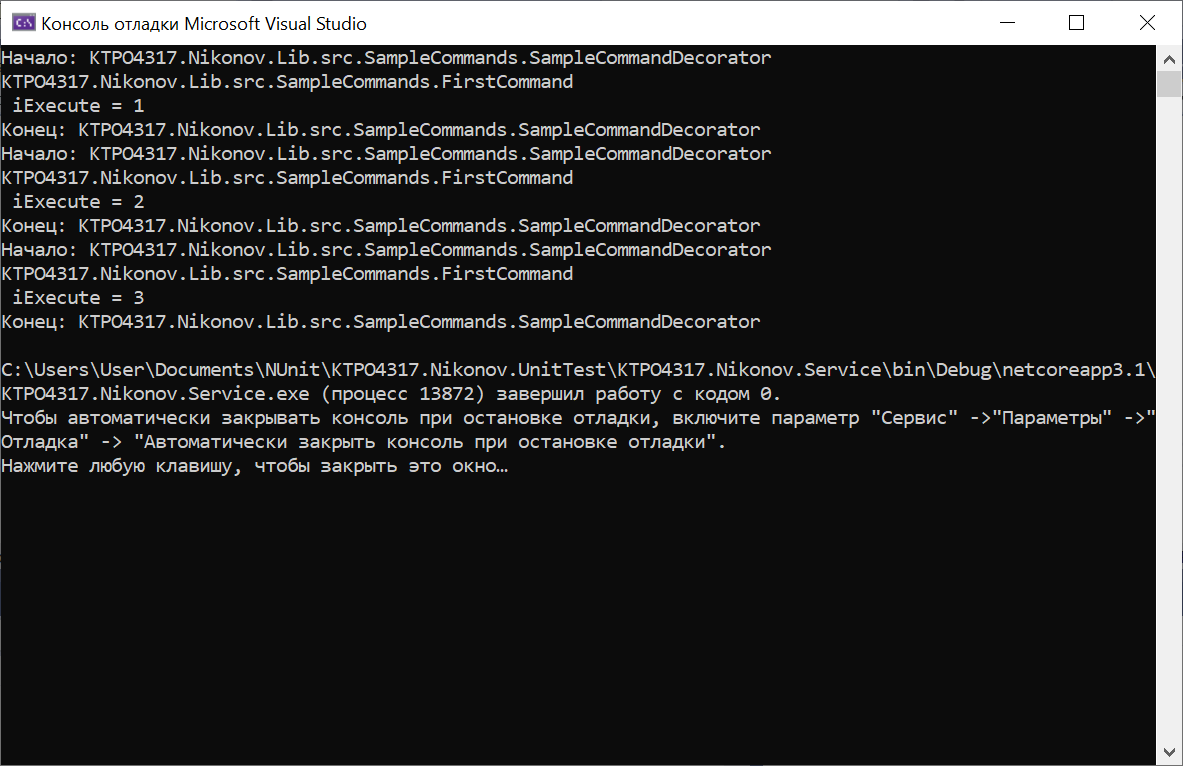


Добавьте функциональность декоратора: пусть он выводит текст «Начало» до вызова декорируемого объекта, и «Конец» - после. Нужно в него внедрить через конструктор представления IView.



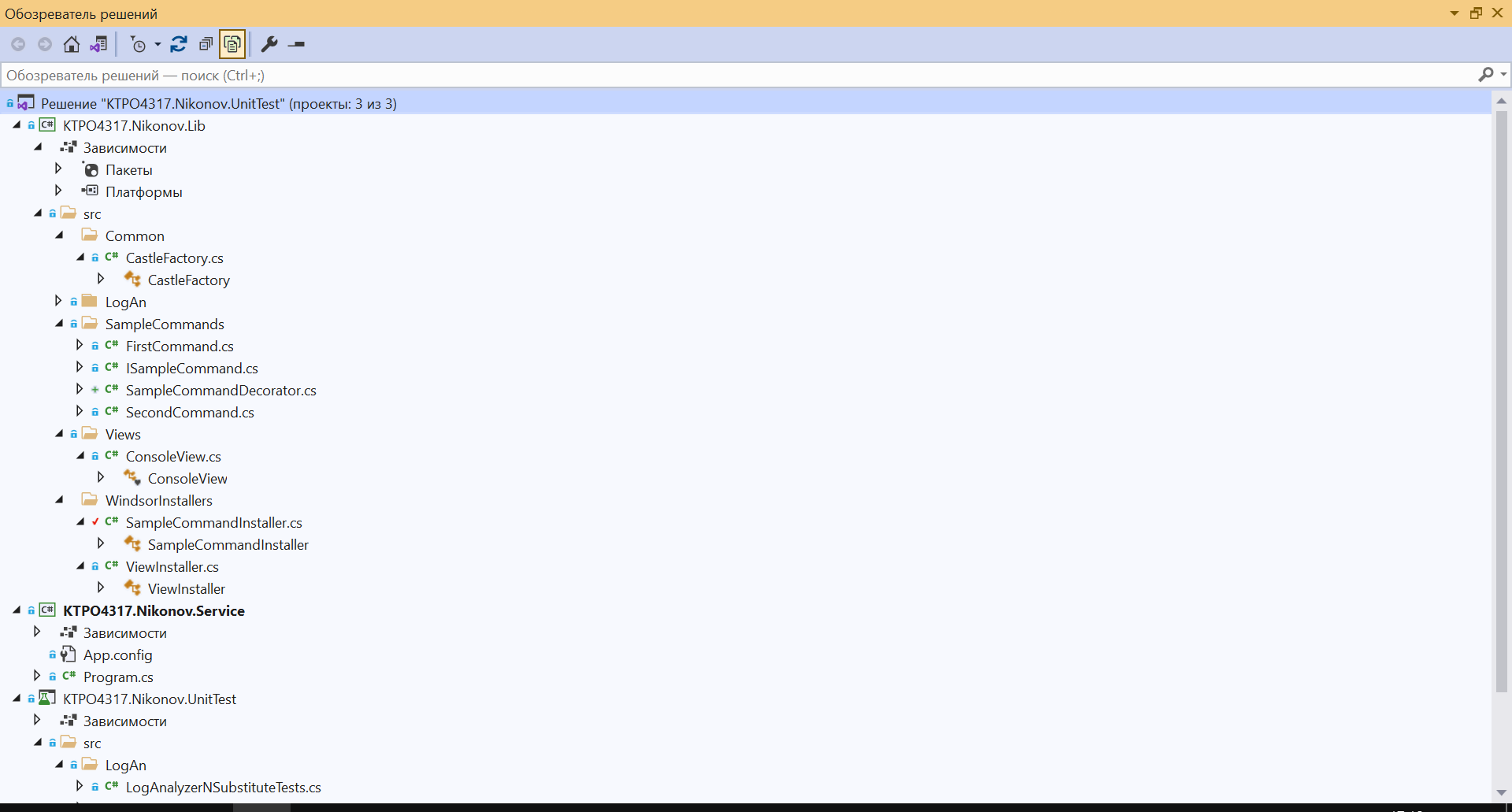
Сконфигурируйте зависимости команд, добавив регистрацию декоратора.





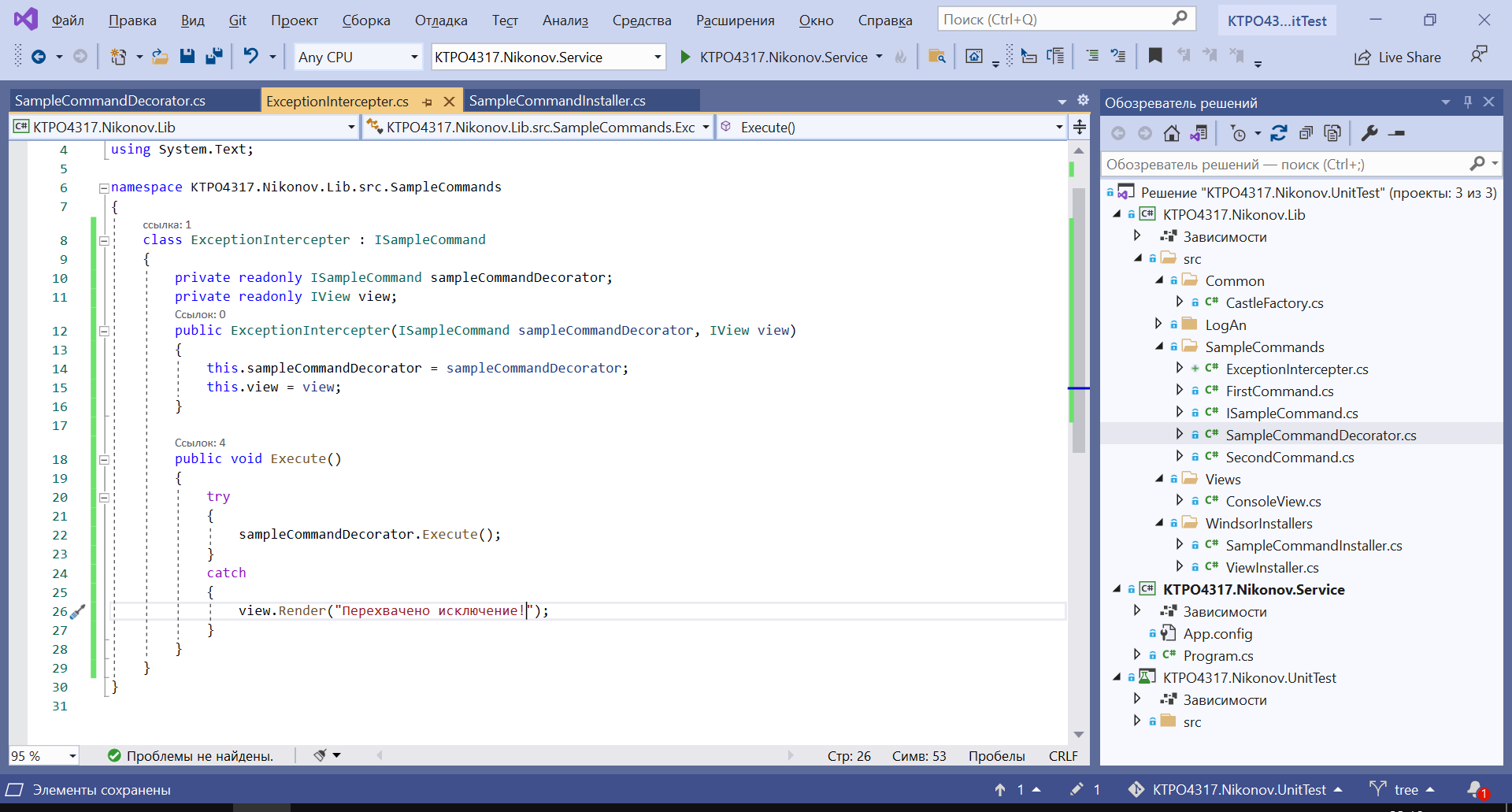
**6. Добавление декоратора для перехвата исключений**

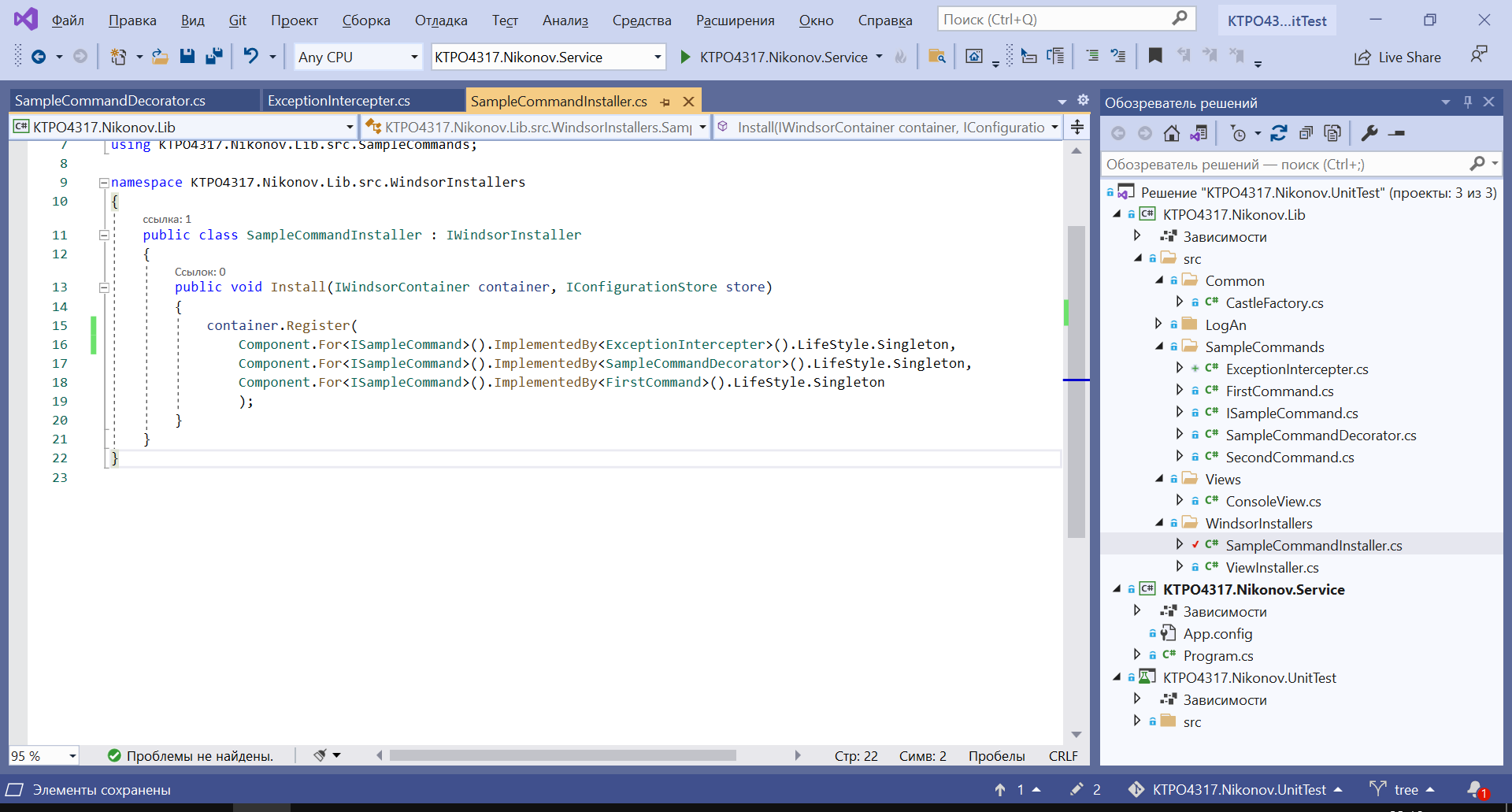
Самостоятельно реализуйте еще один «декоратор», задаче которого будет перехват исключений из декорируемого объекта и вывод текста сообщения на экран. По порядку вызова он должен быть после SampleCommandDecorator.



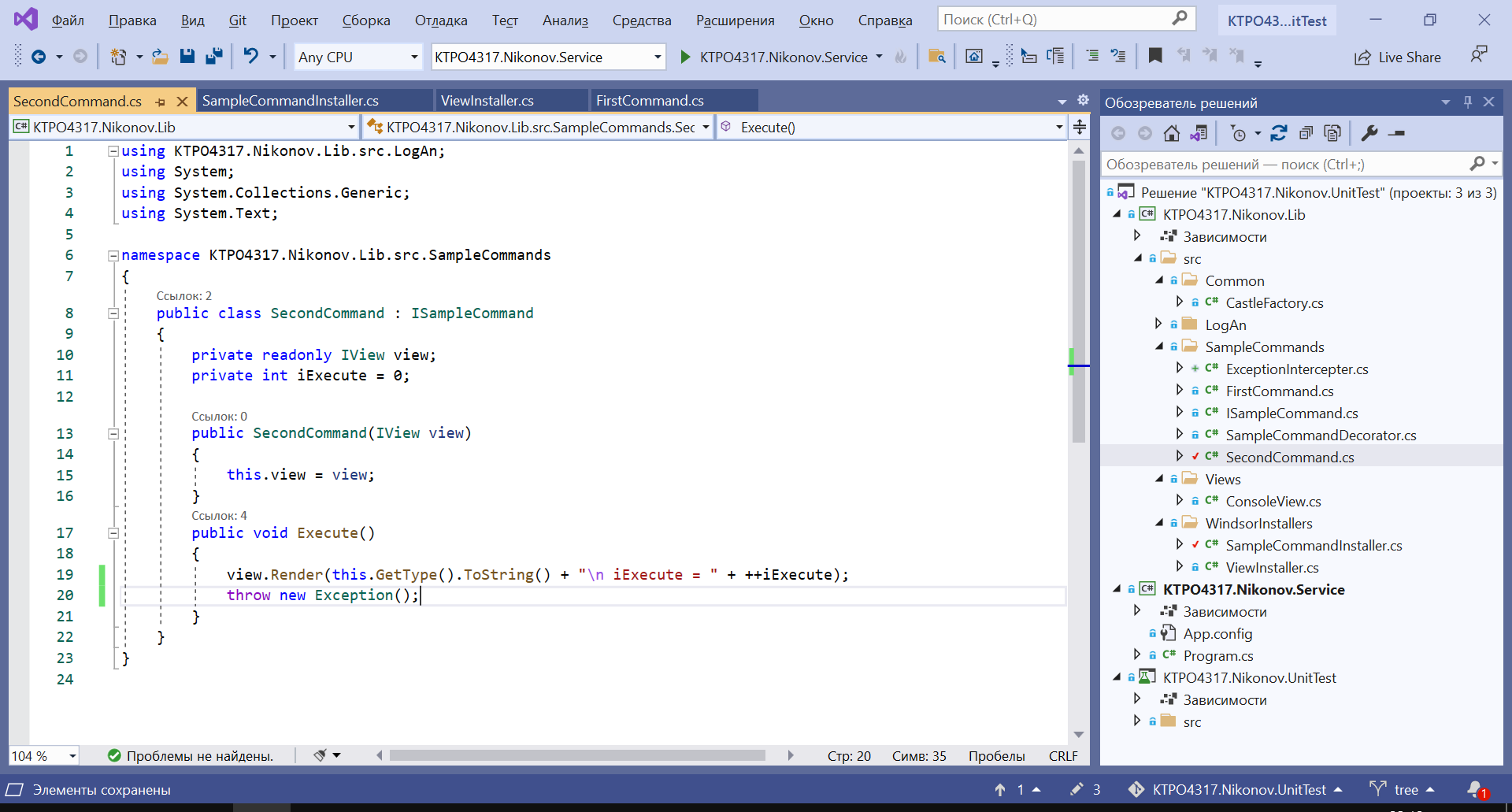
**6. Добавление декоратора для перехвата исключений**

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.

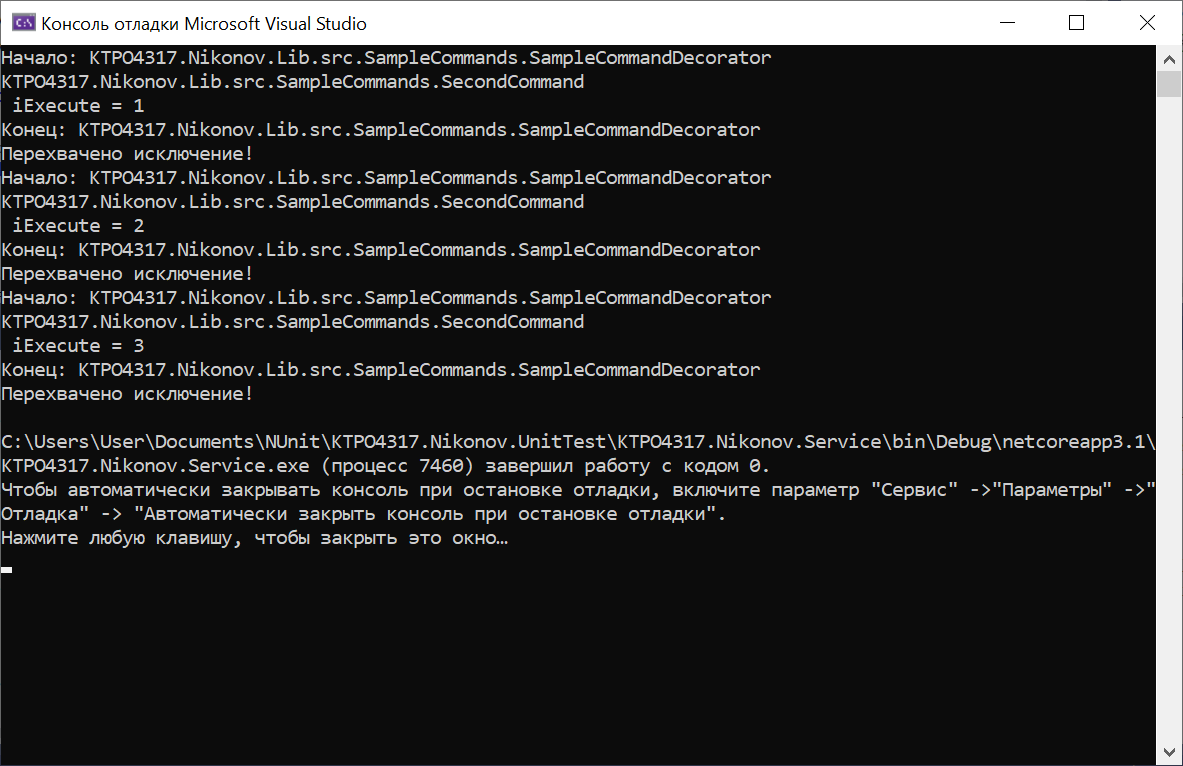




Проведите эксперимент: добавьте в декорируемую команду SecondCommand вызов исключения. Сконфигурируйте зависимости, чтобы использовался этот класс.

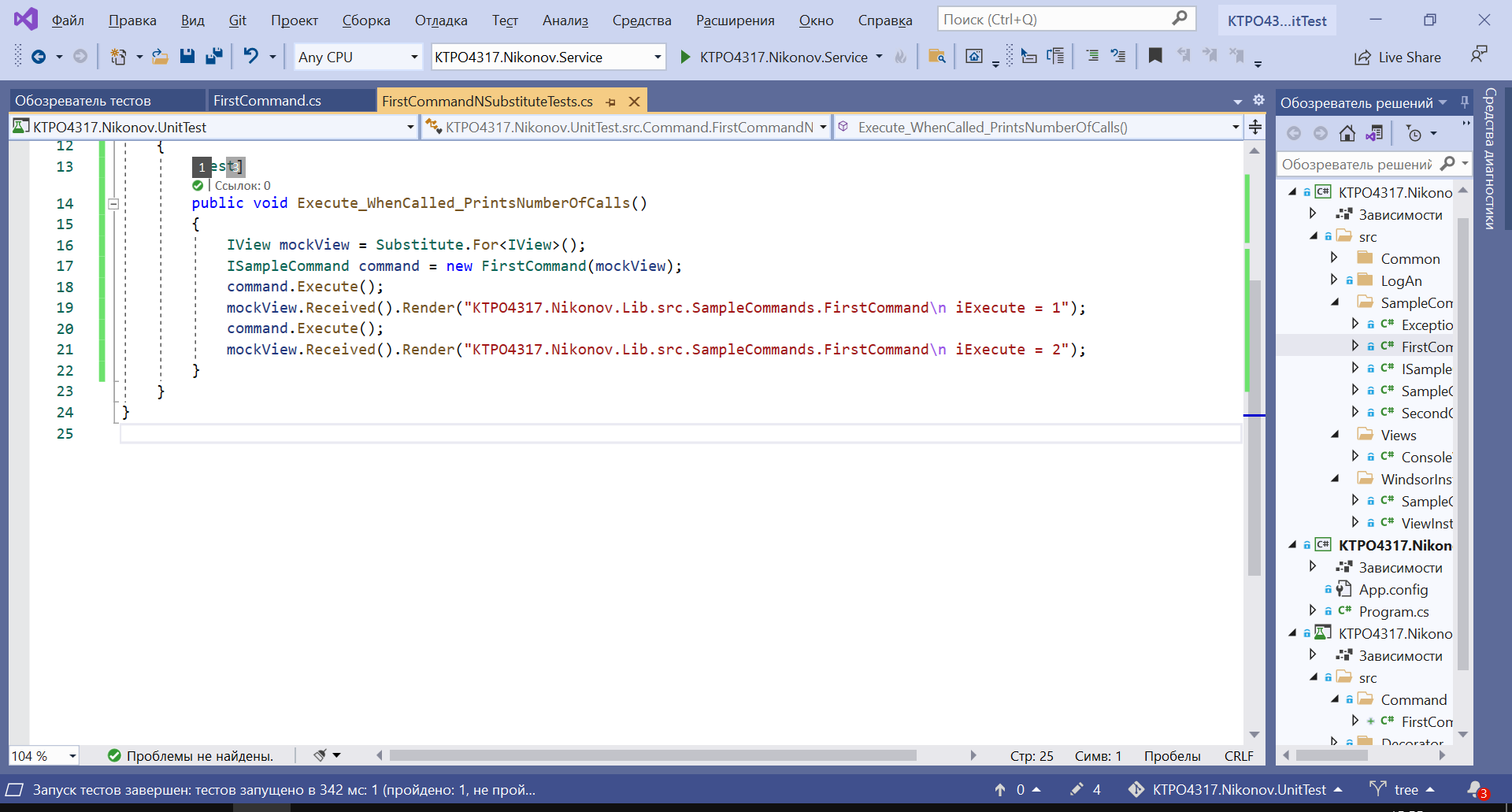


Соберите решение и запустите проект «.Service».Зафиксируйте окно результат выполнения программы.

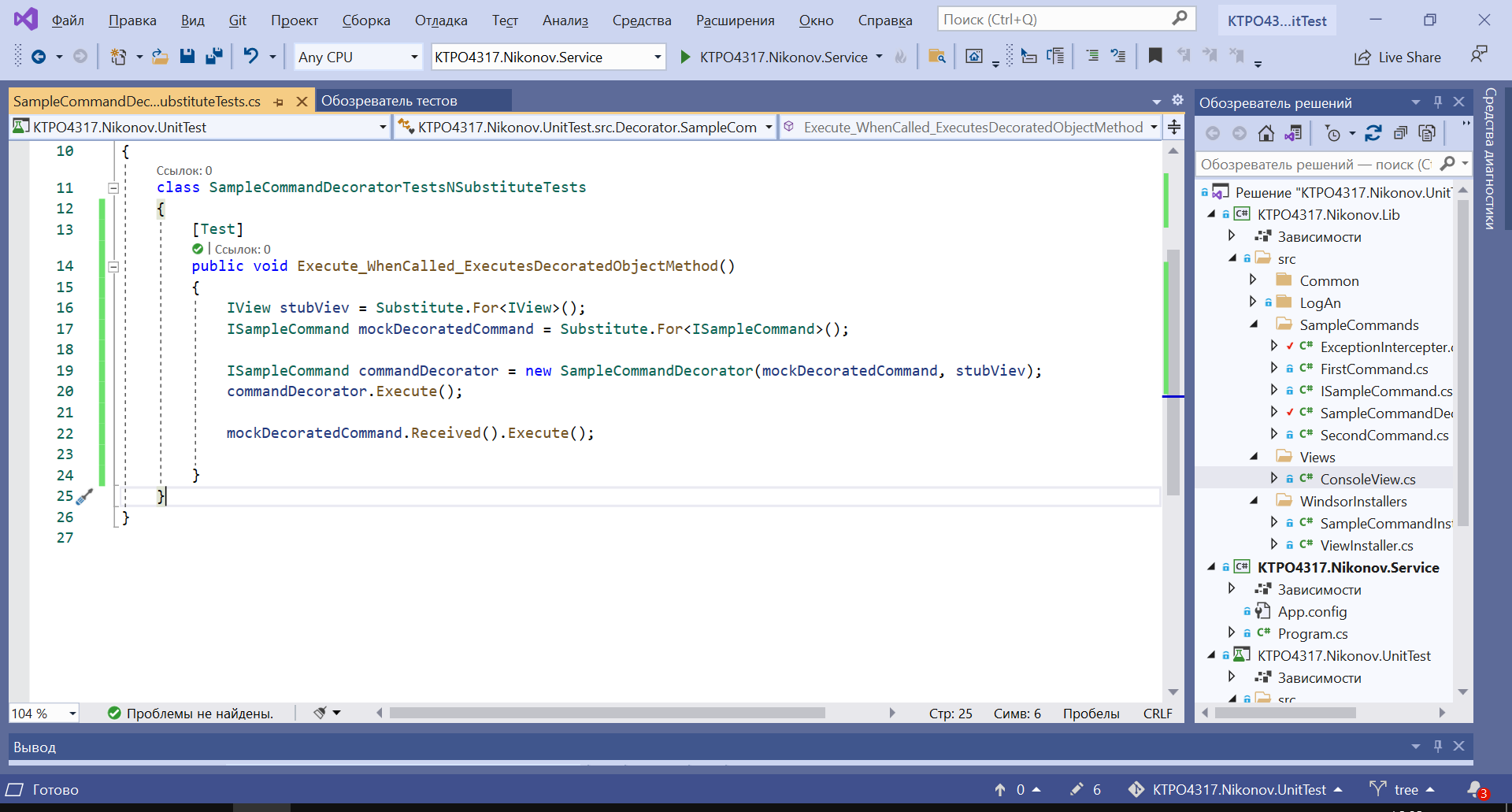


**7. Реализация автономных тестов для разработанных классов**

1. Автономный тест для «команды» FirstCommand: метод Execute() вызывает вывод текста согласно заданию.



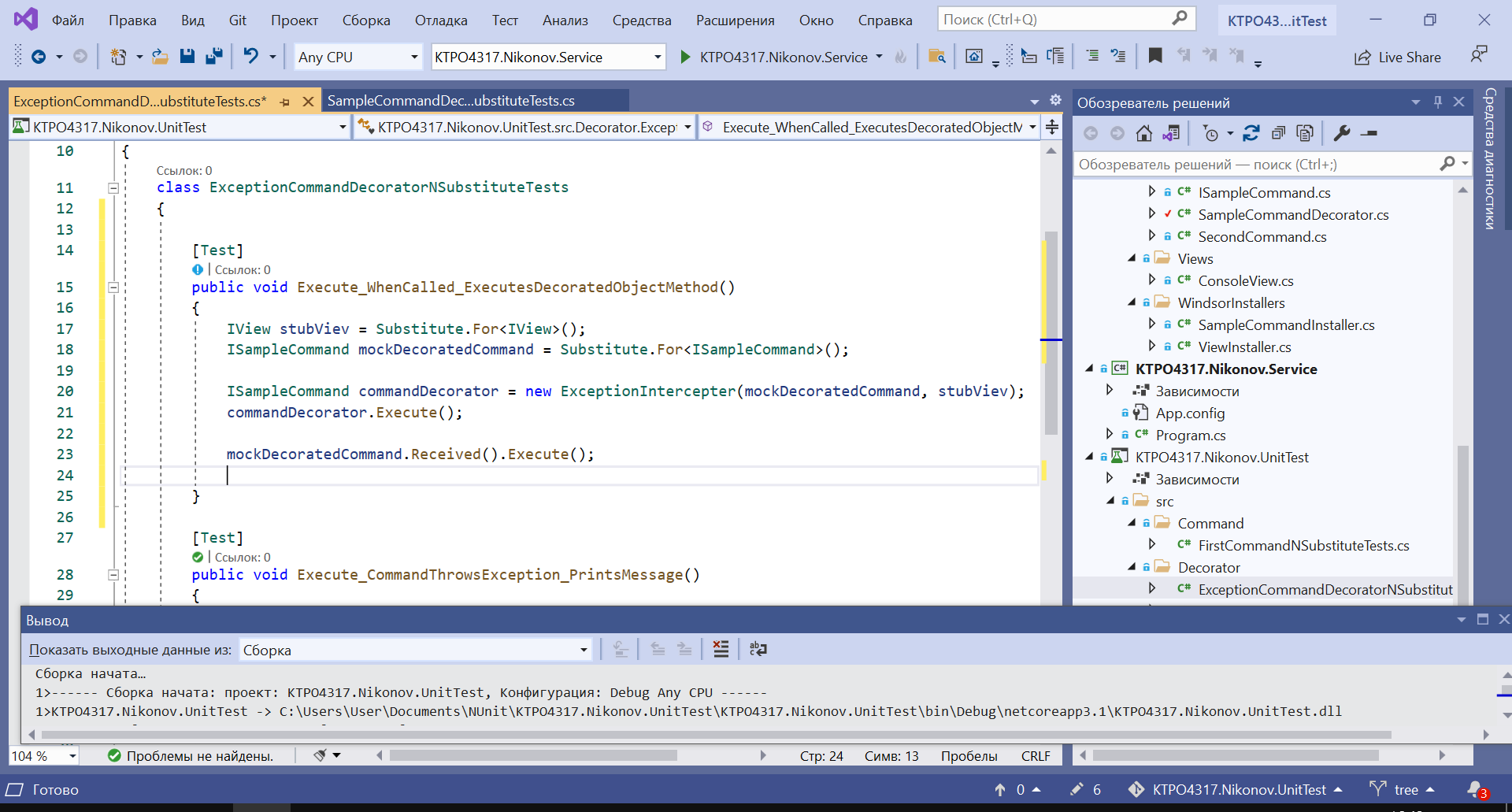
2. Автономный тест для «декоратора» SampleCommandDecorator: метод Execute() вызывает метод декорируемого объекта.

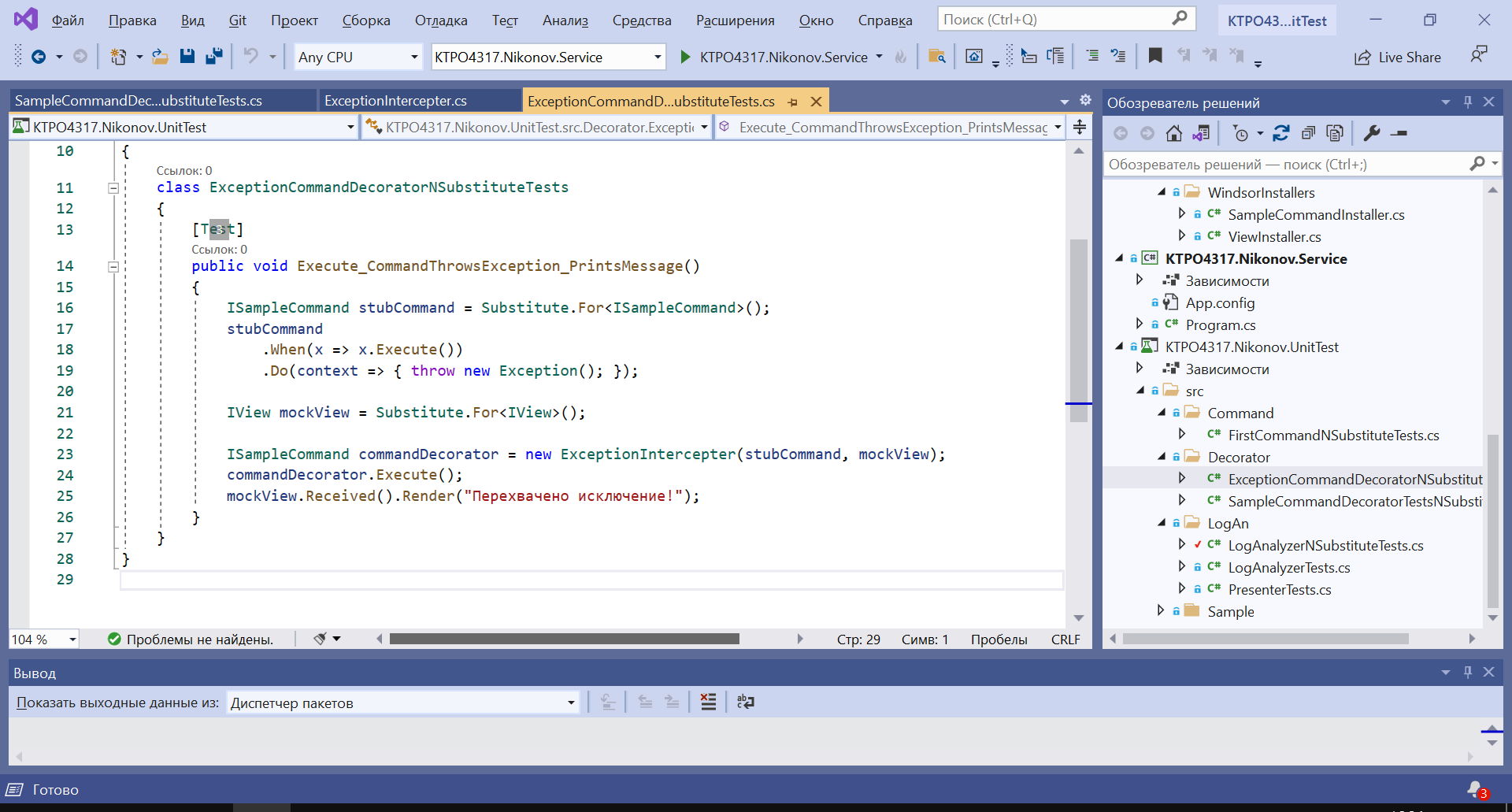


3. Автономный тест для «декоратора» SampleCommandDecorator: метод Execute() вызывает вывод текста согласно заданию.



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. 

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

