Юнит-тестирование в C#

1. Что такое unit-тест

Unit-тест — это автоматизированная проверка одного логического юнита кода в изоляции от остальной системы.

Unit-тест отвечает на вопросы:

- Что произойдёт при корректных данных?
- Что произойдёт при некорректных данных?
- Как код реагирует на граничные случаи?
- Гарантирует ли код сохранение инвариантов?

Хороший unit-тест проверяет:

- **один сценарий поведения**;
- **одну логическую ответственность**;
- **один ожидаемый результат**.

Unit-тесты позволяют бороться с регрессией кода.

Регрессия — это появление ошибок в ранее корректно работавшем коде после внесения изменений.

Unit-тесты, позволяют автоматически обнаруживают регрессии, безопасно рефакторить код и снижают страх перед изменениями.

Без unit-тестов каждое изменение — это риск, приходится проверять вручную весь код, из-за чего ошибки обнаруживаются поздно.

Изоляция как ключевой принцип

Также Unit-тесты помогают нам оценить качество архитектуры системы. Код, который трудно протестировать, почти всегда плохо спроектирован. Когда разработчик пытается написать unit-тест и сталкивается с такими проблемами как:

- невозможность изолировать логику;
- слишком много зависимостей;
- требуется файловая система или консоль,

— это сигнал проблемы в архитектуре приложения, а не проблемы тестов.

Unit-тесты:

- поощряют разделение ответственностей;
- заставляют вводить интерфейсы;
- уменьшают связанность кода.

Unit-тест **обязан быть изолированным**. Это означает, что:

- нет обращения к файловой системе;
- нет работы с консолью;
- нет реальной базы данных;
- нет сетевых запросов.

Причины:

- тесты должны быть быстрыми;
- результат должен быть детерминированным;
- тест не должен зависеть от окружения.

Если тест зависит от внешних факторов — это уже не unit-тест.

Тестируемый юнит

Термин **unit** не имеет строго фиксированного размера.
Юнит — это **минимальная логическая единица поведения**, которую имеет смысл тестировать изолированно. Юнитом может быть отдельный метод или класс.

Метод может считаться самостоятельным юнитом, если он:

- содержит бизнес-логику;
- является детерминированным;
- не зависит от внешнего состояния;
- легко изолируется.

Не имеет смысла писать отдельные unit-тесты для:

- приватных методов без самостоятельной логики;
- геттеров и сеттеров;
- методов-обёрток без поведения.

Unit-тесты для класса стоит писать, если он:

- имеет одну чёткую роль;
- не зависит напрямую от инфраструктуры;
- получает зависимости через конструктор.

Unit-тесты должны:

- работать с публичными методами;
- не обращаться к приватной реализации;
- не зависеть от внутренней структуры класса.

2. Тестовый фреймворк xUnit

Юнит-тесты в .NET невозможно писать «в вакууме» — для этого нужен тестовый фреймворк.
В экосистеме .NET таких фреймворков несколько, но в этом курсе мы будем использовать **xUnit**, так как он отражает современный подход к тестированию и широко применяется в реальных проектах.

На самом базовом уровне unit-тест — это обычный C#-код. Однако без фреймворка:

- тесты невозможно автоматически обнаружить;
- невозможно удобно запускать их пакетно;
- нет стандартных механизмов проверок (assert);
- IDE не понимает, что перед ней тест.

Тестовый фреймворк решает эти задачи:

- помечает методы как тесты;
- управляет жизненным циклом тестов;
- предоставляет API для проверок;
- интегрируется с Visual Studio, Rider и `dotnet test`.

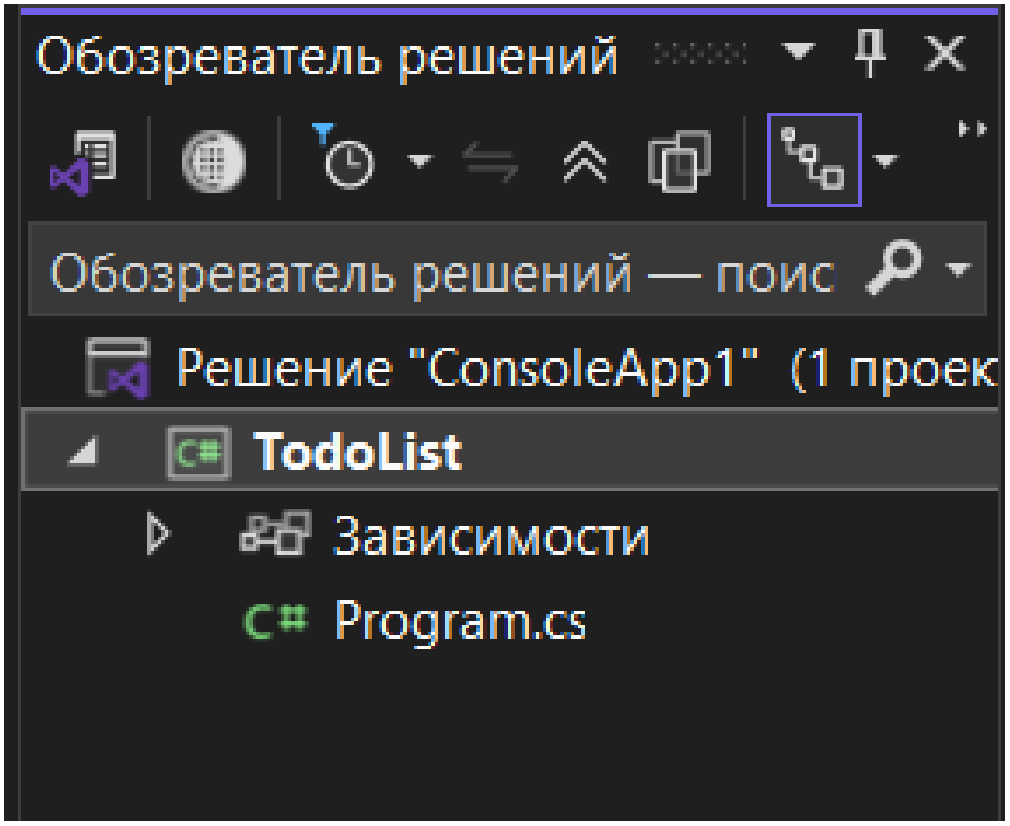
xUnit — это именно такой инфраструктурный слой.

Создание тестового проекта с xUnit

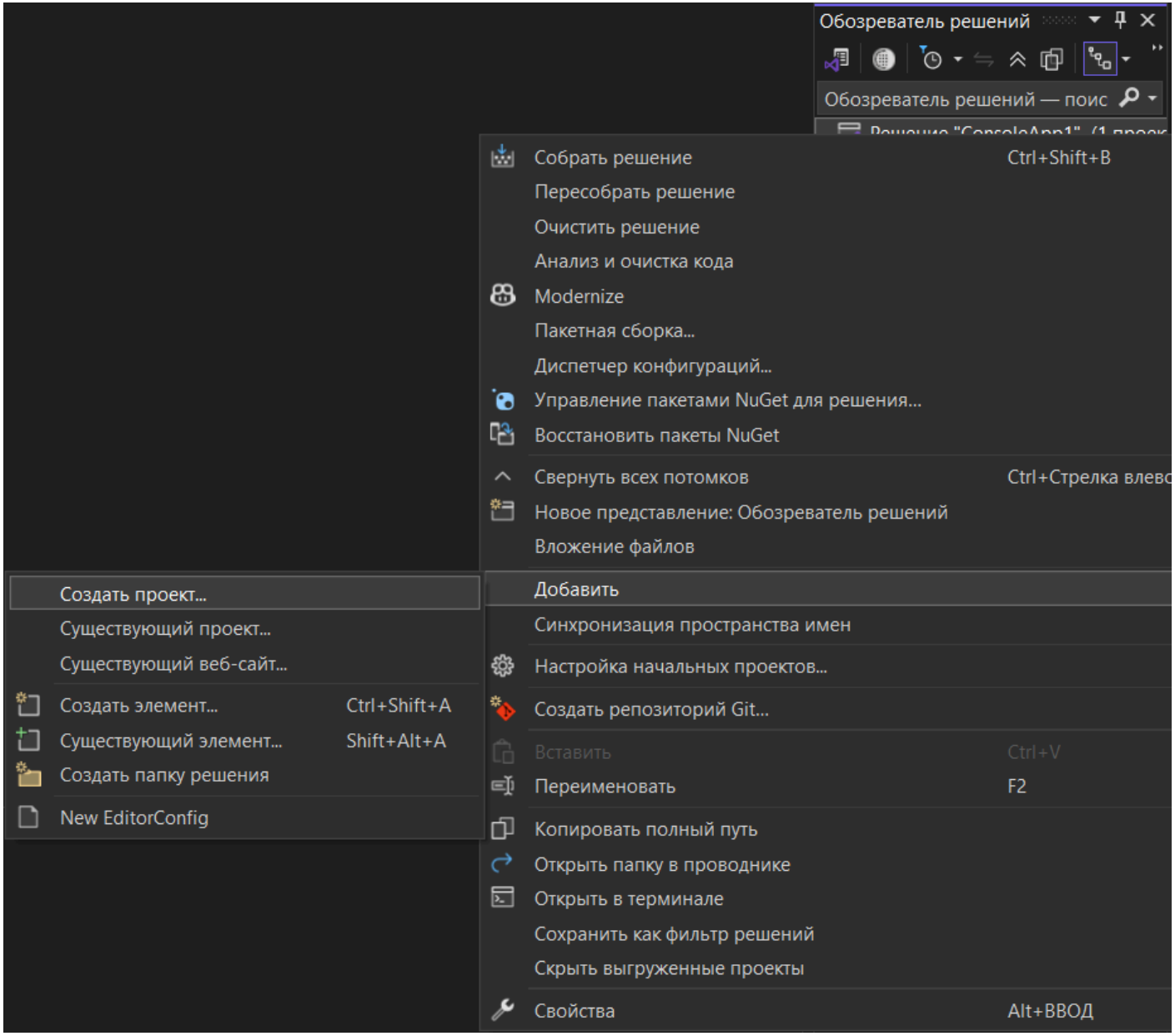
В реальном .NET-проекте unit-тесты всегда выносятся в **отдельный проект**.
Для консольного TodoList это будет, например:

```
TodoList
TodoList.Tests
```

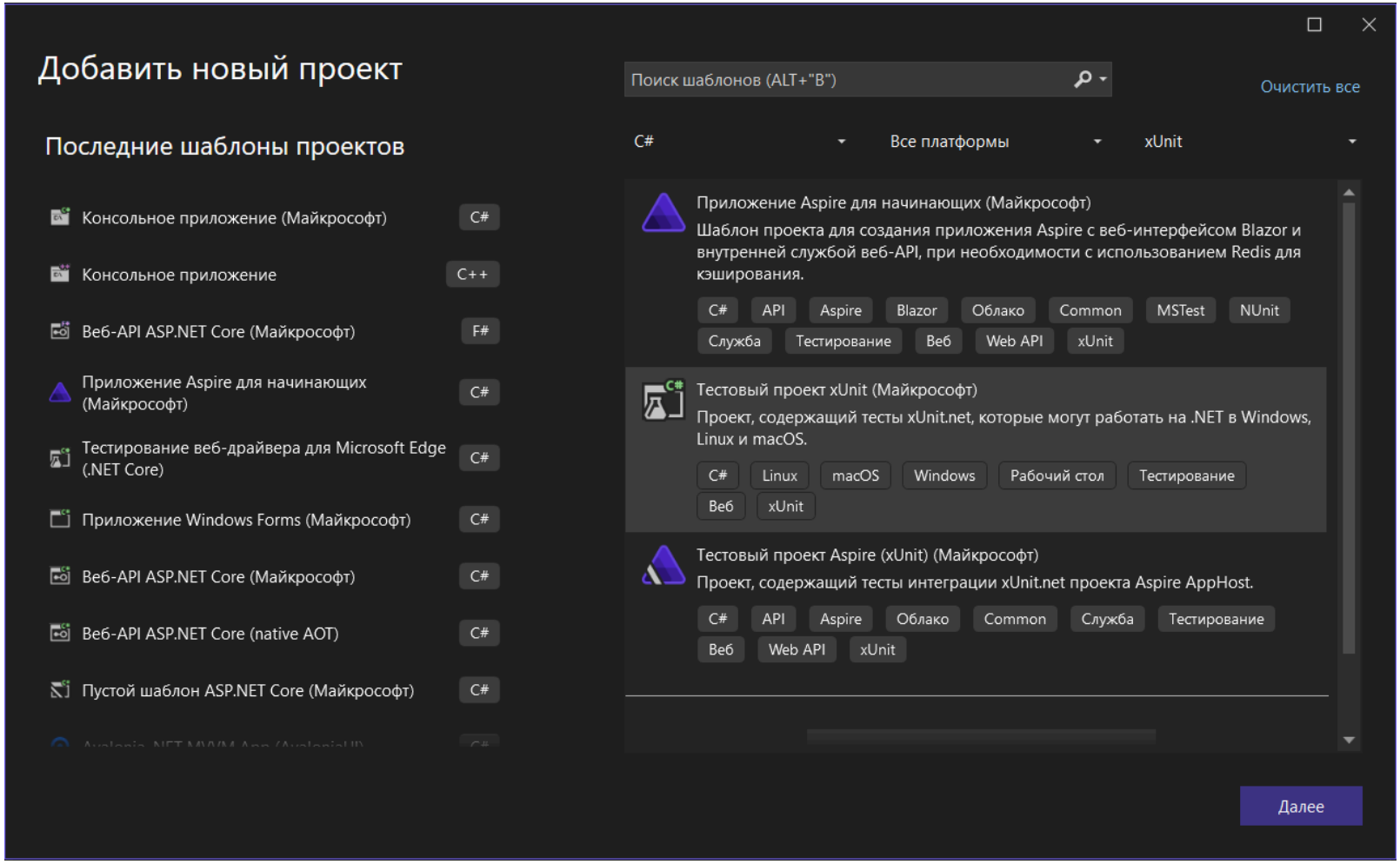
Его можно создать используя IDE.



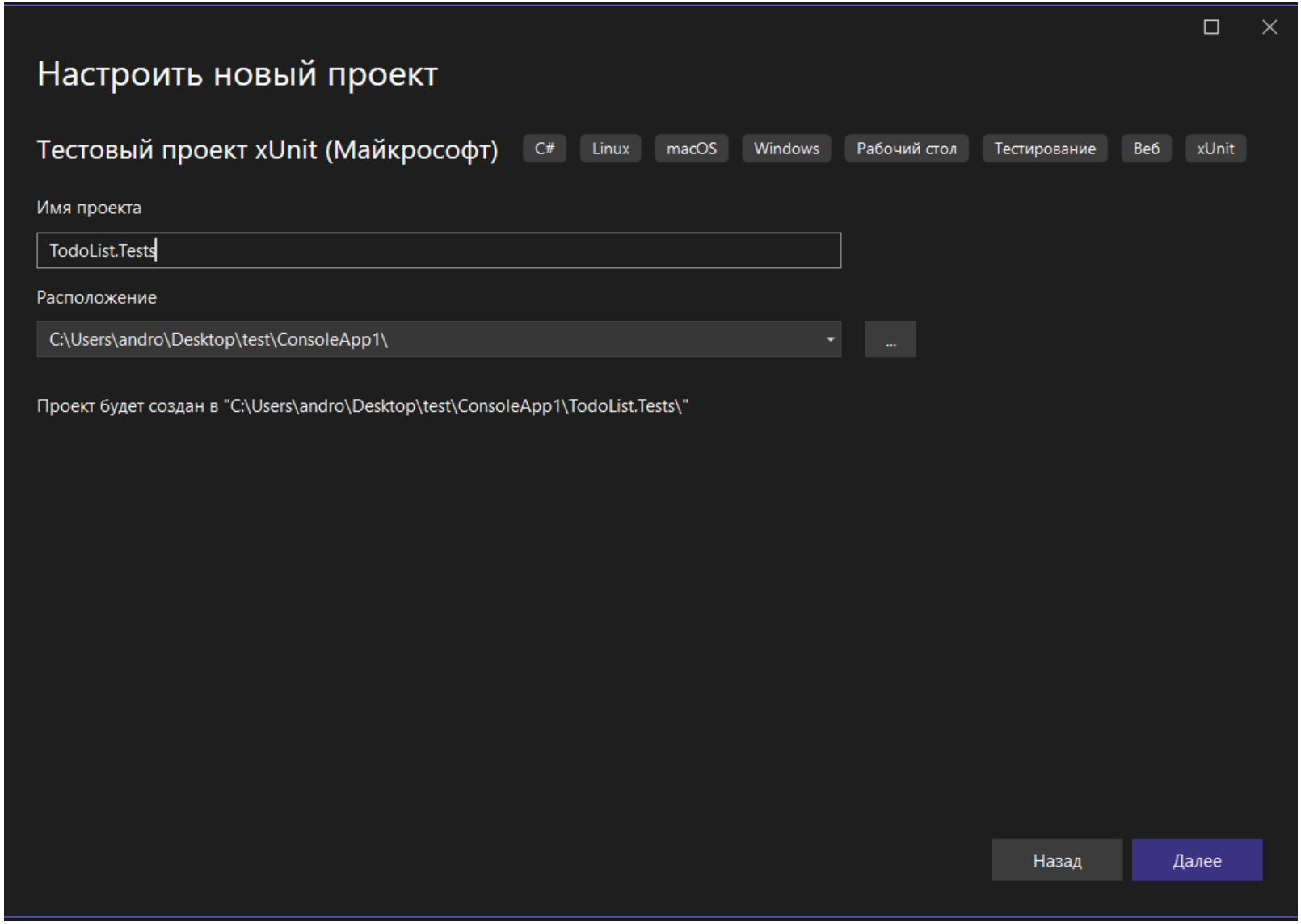
Нажать ПКМ по решению >> `Добавить` >> `Создать проект`



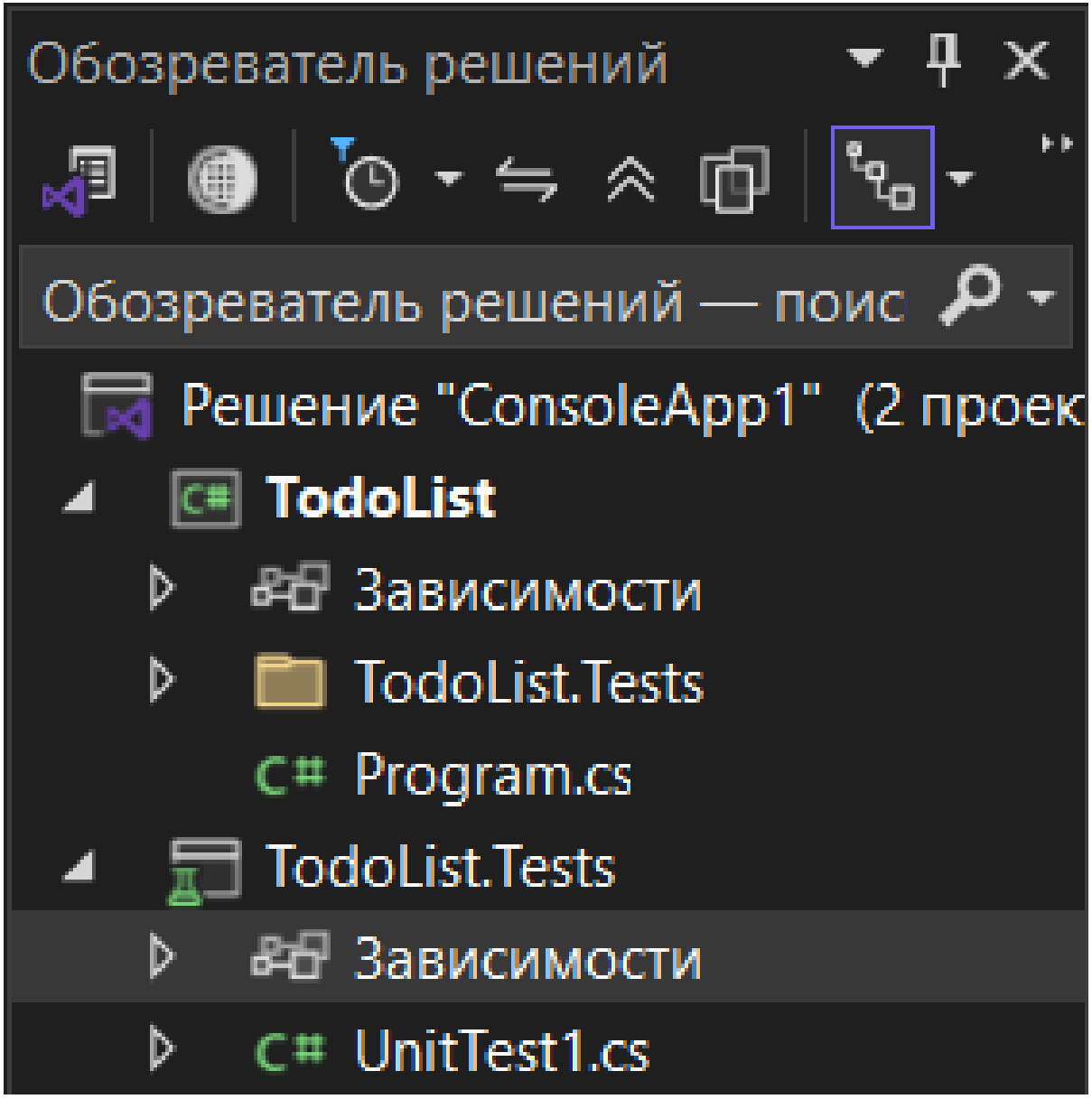
И в шаблонах выбрать **Тестовый проект xUnit**



Выберите нужную папку, куда вы поместите тестовый проект и дайте ему соответствующие название.

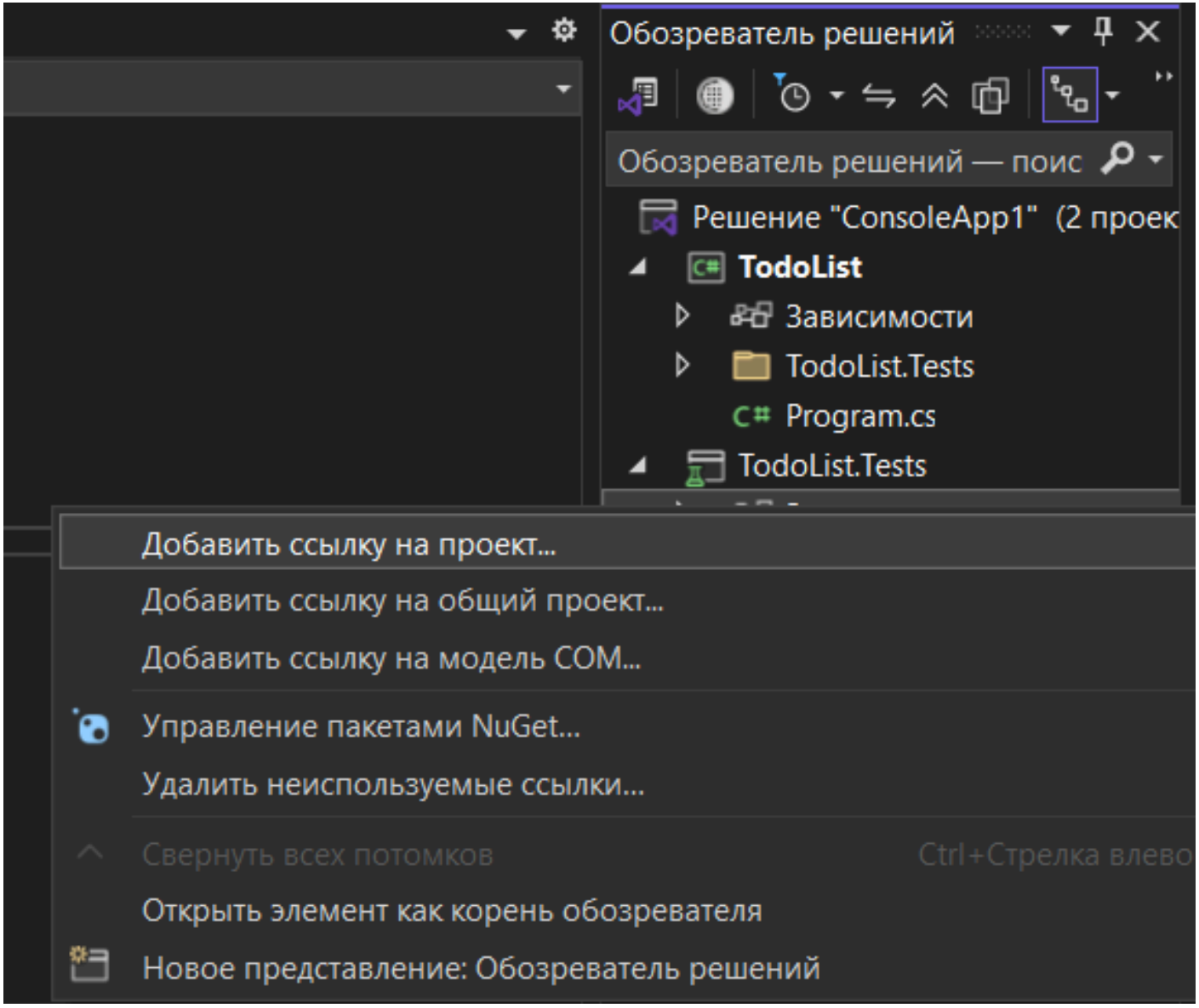


Тестовый проект появится в вашем обозревателе решений.

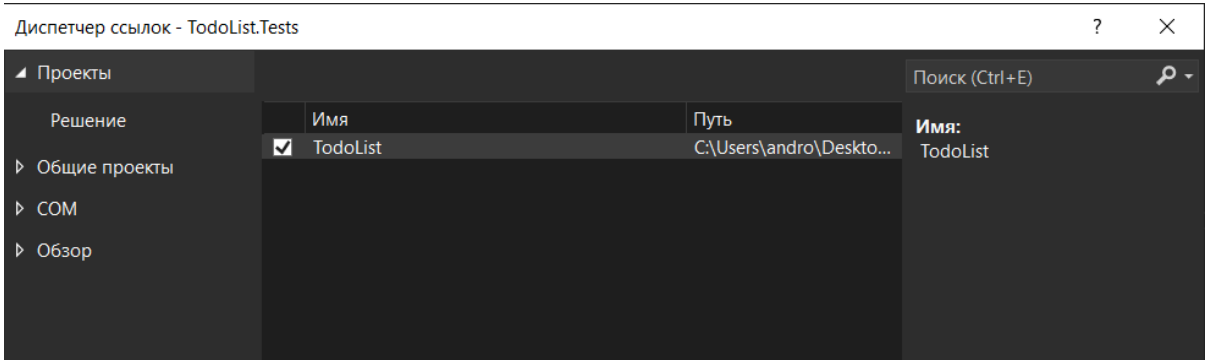


Теперь необходимо добавить в зависимости тестового проекта основной проект.

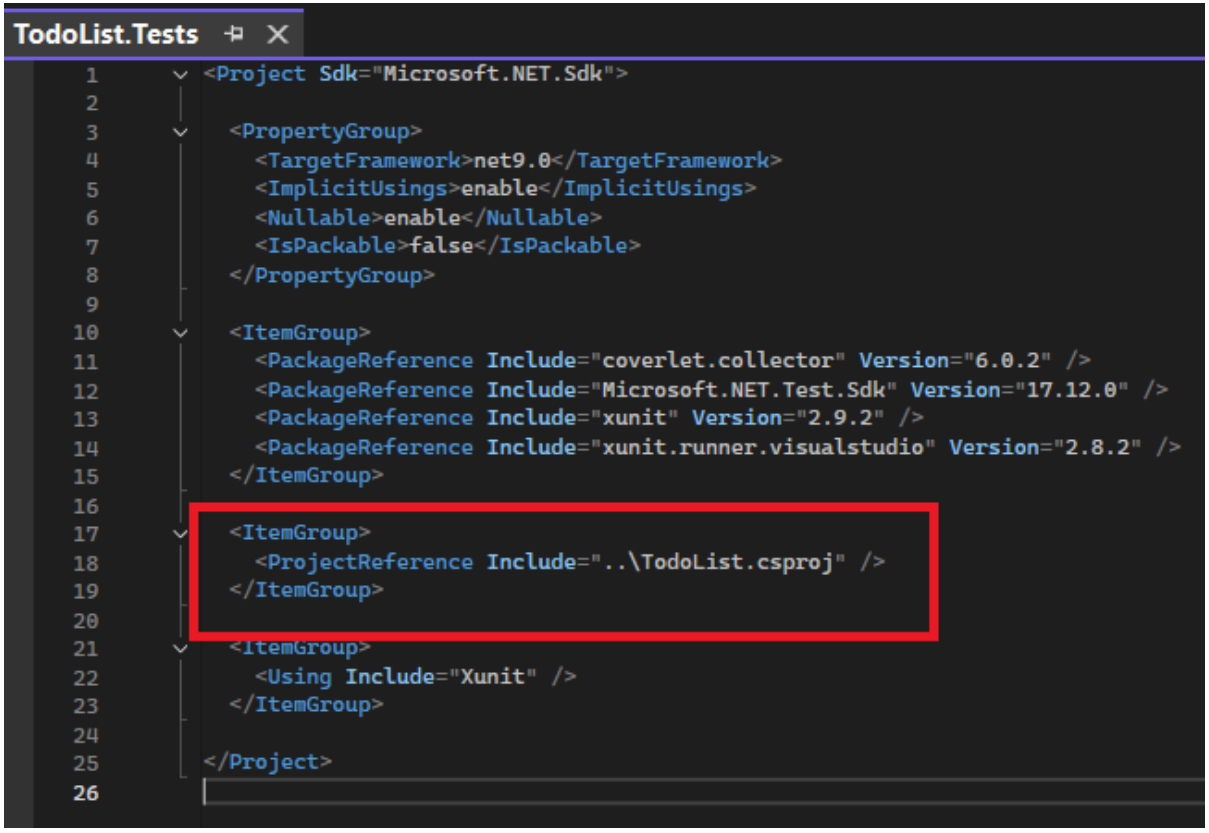
Нажмите ПКМ по разделу `Зависимости` в тестовом проекте и выберите `Добавить ссылку на проект`.



Поставьте галочку на против нужного проекта.



В файле тестового проекта `TodoList.Tests.csproj` появится ссылка на основной проект



Структура unit-теста

После создания тестового, давайте рассмотрим минимальный тест, для того чтобы понять, как они устроены:

```
using Xunit;

public class SampleTests
{
    [Fact]
    public void TestExample()
    {
        Assert.True(true);
    }
}
```

Этот код уже является корректным unit-тестом:

- метод помечен атрибутом `[Fact]`;
- внутри есть проверка (`Assert`);
- тест будет найден и запущен автоматически.

В xUnit **[Fact]** означает, что это самостоятельный тест без входных параметров. xUnit больше не требует никаких дополнительных настроек — достаточно атрибута `[Fact]`.

Assert в xUnit

`Assert` — это набор статических методов для проверки ожиданий.
Они формируют «язык утверждений», на котором пишутся тесты.

Примеры типичных проверок:

```
Assert.Equal(expected, actual);
Assert.True(condition);
Assert.False(condition);
Assert.Null(value);
Assert.NotNull(value);
```

Для коллекций:

```
Assert.Empty(items);
Assert.Single(items);
Assert.Contains(item, items);
```

Важно понимать:
Assert — это не просто проверка, это формулировка ожидания.

Если Assert падает, тест считается проваленным.

Как запустить тесты

Каждый тест в xUnit:

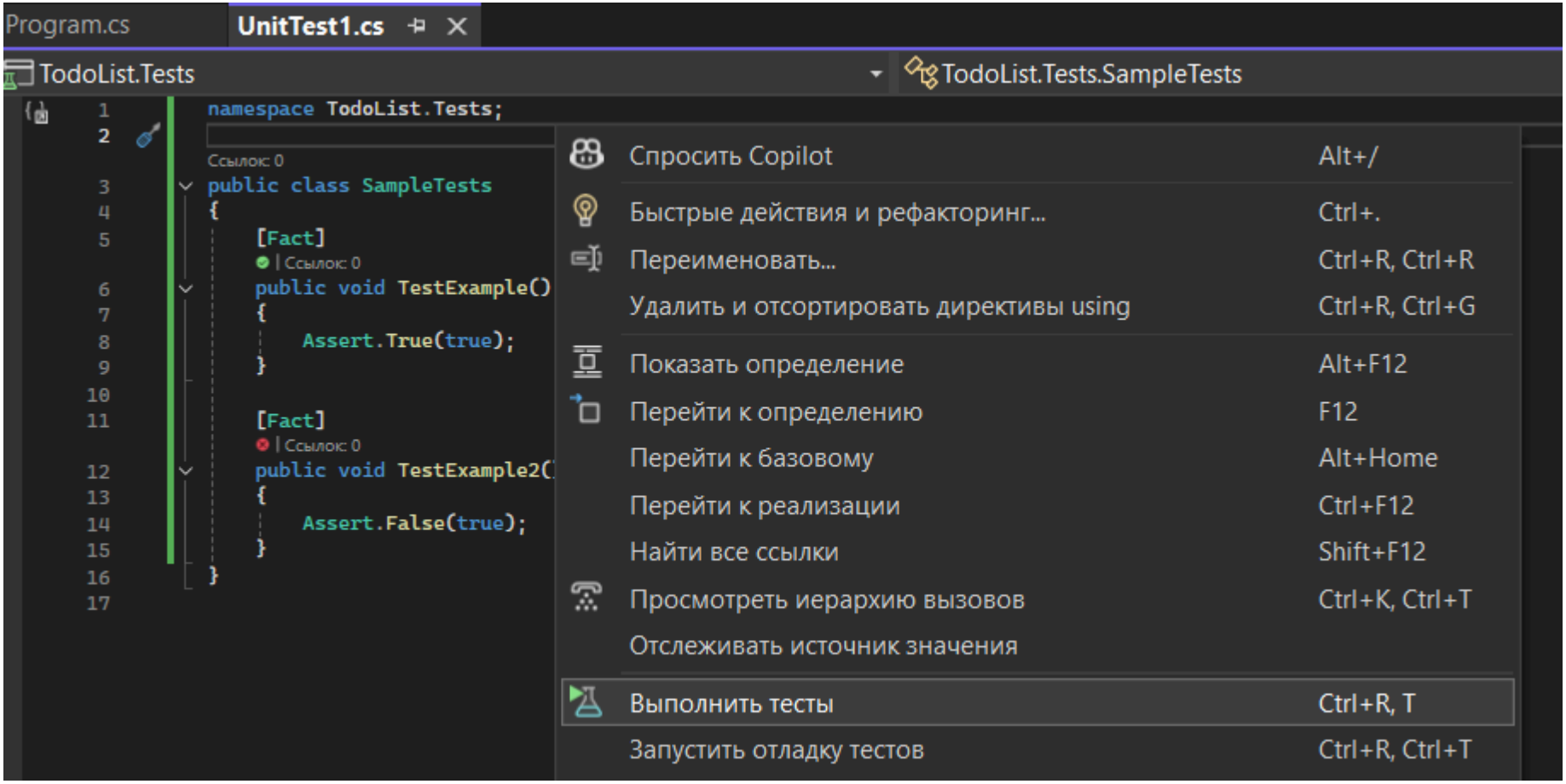
- выполняется независимо;
- запускается в новом экземпляре тестового класса;
- не знает о состоянии других тестов.

Тесты можно запускать:

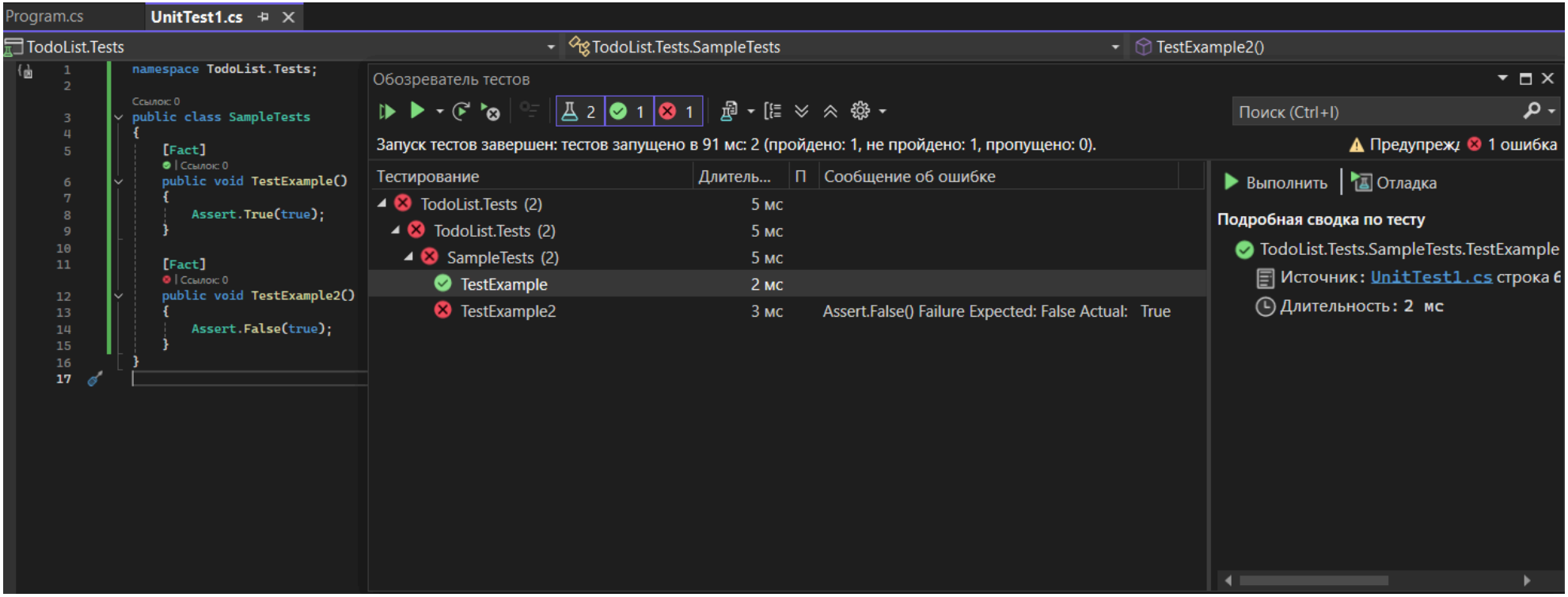
- через Test Explorer (Обозреватель тестов) в Visual Studio;
- через встроенный runner в Rider;
- через команду:

```
dotnet test
```

Пример запуска теста в VisualStudio:



После запуска тестов появится окно обозревателя тестов, где вы сможете посмотреть какие тесты были запущены, сколько тестов было пройдено успешно, а сколько провалено и почему. Так же в этом окне можно заново запустить все или некоторые тесты. После запуска, над тестами в коде появятся соответствующие значки.



3. Структура unit-теста

Важно помнить, что unit-тест — это не просто метод с проверкой. Это маленький сценарий, у которого есть начало, действие и ожидаемый результат. Если этот сценарий не выражен явно, тест перестаёт быть полезным.

Любой корректный unit-тест отвечает на вопрос:

> «Если система находится в определённом состоянии и происходит конкретное действие, то результат должен быть таким».

Это утверждение всегда можно мысленно разделить на три части:

1. подготовка состояния;
2. выполнение действия;
3. проверка результата.

Даже если разработчик не осознаёт этого, хороший тест **всегда** содержит эти три этапа. В xUnit они не задаются явно, но должны быть видны из структуры кода.

Паттерн Arrange / Act / Assert

Рассмотрим простой пример из TodoList — добавление задачи.

```
[Fact]
public void AddTask_WithValidTitle_TaskIsAdded()
{
    var service = new TaskService();
    service.AddTask("Buy milk");
    Assert.Single(service.Tasks);
}
```

Этот тест корректен, но пока плохо читается. Чтобы понять его смысл, нужно анализировать код целиком. Теперь перепишем его, сделав структуру явной:

```
[Fact]
public void AddTask_WithValidTitle_TaskIsAdded()
{
    // Arrange
    var service = new TaskService();

    // Act
    service.AddTask("Buy milk");

    // Assert
    Assert.Single(service.Tasks);
}
```

Функционально тест не изменился, но его структура стала очевидной даже для человека, который видит этот код впервые.

Это пример **паттерна Arrange / Act / Assert**.

AAA описывает базовую причинно-следственную модель:

> Если система находится в определённом состоянии (Arrange)
> и над ней совершается одно действие (Act),
> то результат должен соответствовать ожиданию (Assert).

Эта модель совпадает с тем, как мы рассуждаем о корректности кода.
Паттерн просто делает это рассуждение явным и проверяемым.

Если тест невозможно разложить на Arrange, Act и Assert, значит либо тест, либо код спроектированы неправильно.

Arrange: подготовка состояния

Секция **Arrange** отвечает на вопрос:

> «В каком состоянии находится система перед действием?»

Здесь создаются:

- объекты, которые будут тестироваться;
- входные данные;
- начальное состояние.

Пример с более сложной логикой:

```
var service = new TaskService();
service.AddTask("Task 1");
service.AddTask("Task 2");
```

Важно помнить: если подготовка занимает слишком много строк или выглядит сложно, это сигнал, что тестируемый код перегружен.

Act: одно действие

Секция **Act** должна содержать **ровно одно логическое действие**.
Это ключевое правило.

```
service.CompleteTask(1);
```

Если в секции Act несколько действий, тест начинает проверять сразу несколько сценариев, и его смысл размывается.

Плохой пример:

```
service.AddTask("Task");
service.CompleteTask(1);
service.DeleteTask(1);
```

Хороший unit-тест всегда проверяет **одну причину и одно следствие**.

Assert: проверка результата

Секция **Assert** формулирует ожидание от системы.
Это самая важная часть теста.

```
Assert.True(service.Tasks[0].IsCompleted);
```

Assert должен быть:

- конкретным;
- однозначным;
- легко интерпретируемым.

Если тест падает, разработчик должен сразу понимать, **что именно пошло не так**.

Сколько Assert должно быть в тесте

Формально xUnit не ограничивает количество `Assert`.
Практически правило такое:

> Один unit-тест — одно логическое ожидание.

Допустимо несколько `Assert`, если они:

- проверяют одно и то же состояние;
- логически связаны.

Пример допустимого варианта:

```
Assert.Equal("Buy milk", task.Title);
Assert.False(task.IsCompleted);
```

Но если `Assert` начинают проверять разные аспекты поведения — тест нужно разделить.

Структура тестового класса

Unit-тесты обычно группируются по тестируемому классу.

```
public class TaskServiceTests
{
    [Fact]
    public void AddTask_WithValidTitle_TaskIsAdded() { }

    [Fact]
    public void CompleteTask_TaskExists_StatusChanged() { }
}
```

Такой подход:

- упрощает навигацию;
- делает тесты предсказуемыми;
- отражает структуру проекта.

4. Параметризованные тесты

На определённом этапе разработчик сталкивается с типичной ситуацией: один и тот же unit-тест приходится писать несколько раз, меняя только входные данные. Логика теста остаётся прежней, но количество тестов растёт, код дублируется, а поддержка становится всё дороже.

Параметризованные тесты решают именно эту проблему. Они позволяют описать **одно поведение** и проверить его **на наборе разных входных данных**, не теряя читаемости и смысла.

Представим, что в TodoList есть логика валидации названия задачи. Название считается некорректным, если оно:

- пустое;
- состоит только из пробелов;
- содержит недопустимые символы.

Наивный подход — написать отдельный `[Fact]` для каждого случая. Такой код быстро разрастается и перестаёт быть удобным. Проблема здесь в том, что все они проверяют одно и то же поведение.

Параметризованный тест отвечает на вопрос:

> Как система ведёт себя **при разных входных данных, но в одном и том же сценарии?**

В xUnit эта идея реализуется через:

- `[Theory]` — параметризованный тест;
- источники данных (`InlineData` , `MemberData` и др.).

Отличие [Fact] и [Theory]

`[Fact]` используется, когда:

- входных данных нет;
- сценарий уникален.
  `[Theory]` используется, когда:
- сценарий один;
- данные разные;
- ожидаемый результат одинаков по смыслу.

Напишем параметризованный тест для валидации названия задачи с использованием `[Theory]`:

```
[Theory]
[InlineData("")]
[InlineData("   ")]
[InlineData("/;")]
public void AddTask_InvalidTitle_ThrowsException(string title)
{
    // Arrange
    var service = new TaskService();

    // Act & Assert
    Assert.Throws<ArgumentException>(() =>
        service.AddTask(title)
    );
}
```

Здесь тест описывает одно поведение, входные данные передаются параметром, а каждая строка `InlineData` — это отдельный запуск теста. Если один из вариантов падает, xUnit покажет, с какими данными тест не прошёл.

Проверка разных результатов

Параметризованные тесты могут проверять не только разные входы, но и разные ожидаемые результаты.

```
[Theory]
[InlineData("Task 1", true)]
[InlineData("", false)]
public void IsValidTitle_ReturnsExpectedResult(string title, bool expected)
{
    // Arrange
    var validator = new TaskValidator();

    // Act
    var result = validator.IsValid(title);

    // Assert
    Assert.Equal(expected, result);
}
```

Такой тест остаётся читаемым, пока:

- количество параметров разумное;
- смысл параметров очевиден.

5. Тестирование исключений

Исключения — это не «аварии», а часть кода. Хорошо спроектированный метод явно определяет, **в каких ситуациях он не может продолжать работу** и какое исключение при этом выбрасывает. Unit-тесты должны проверять исключения так же строго, как и успешные сценарии.

Тестирование исключений — это не попытка «поймать ошибку», а способ зафиксировать ожидаемое поведение системы при некорректных входных данных или состояниях.

Рассмотрим метод добавления задачи:

```
public void AddTask(string title)
{
    if (string.IsNullOrWhiteSpace(title))
        throw new ArgumentException("Title cannot be empty");

    // логика добавления
}
```

Здесь `ArgumentException` — не ошибка реализации, а **явно определённое поведение**, которое необходимо протестировать.

Базовый способ тестирования исключений в xUnit

В xUnit для этого используется `Assert.Throws`.

```
[Fact]
public void AddTask_EmptyTitle_ThrowsArgumentException()
{
    // Arrange
    var service = new TaskService();

    // Act & Assert
    Assert.Throws<ArgumentException>(() =>
        service.AddTask("")
    );
}
```

Обрати внимание:

- тест проверяет **тип исключения**, а не просто факт падения;
- тест остаётся детерминированным;
- никакого `try/catch` внутри теста нет.

Важно помнить, что в тестах нельзя использовать try/catch. Это плохая практика по нескольким причинам:

- тест не проверяет тип исключения;
- тест может «пройти» при любом исключении;
- читаемость резко ухудшается.

`Assert.Throws` — декларативный и безопасный способ выразить намерение теста.

Проверка сообщения исключения

Иногда важно не только исключение, но и его сообщение — например, если оно выводится пользователю.

```
[Fact]
public void AddTask_EmptyTitle_ExceptionHasCorrectMessage()
{
    // Arrange
    var service = new TaskService();

    // Act
    var exception = Assert.Throws<ArgumentException>(() =>
        service.AddTask("")
    );

    // Assert
    Assert.Equal("Title cannot be empty", exception.Message);
}
```

Исключения и параметризованные тесты

Исключения отлично сочетаются с `[Theory]`.

```
[Theory]
[InlineData("")]
[InlineData("   ")]
[InlineData("/;")]
public void AddTask_InvalidTitles_ThrowsArgumentException(string title)
{
    // Arrange
    var service = new TaskService();
```

```
    // Act & Assert
    Assert.Throws<ArgumentException>(() =>
        service.AddTask(title)
    );
}
```

Такой тест сразу фиксирует весь класс некорректных данных и делает контракт явным.

6. Именование unit-тестов

Unit-тест — это не просто проверка кода. Это документ, который объясняет, как система должна себя вести. В хорошо спроектированном проекте тесты читаются как спецификация поведения, а их имена позволяют понять смысл теста без открытия кода.

Важно думать об именах тестов, потому что в реальном проекте разработчик чаще:

- читает имена тестов;
- смотрит отчёт о падениях;
- анализирует, какой сценарий сломался,

чем пишет сами тесты.

Когда тест падает, первое, что мы видим — **его имя**.
Именно оно должно сразу ответить на вопрос:

> Что пошло не так и в каком сценарии?

Плохие имена тестов

Рассмотрим несколько типичных примеров:

```
[Fact]
public void Test1()
```

```
[Fact]
public void AddTaskTest()
```

```
[Fact]
public void CheckAddTask()
```

Все эти имена бесполезны. Они:

- не описывают условия;
- не описывают ожидаемое поведение;
- не помогают при падении теста.

Если такой тест упал, разработчику всё равно придётся открывать код.

Хорошее имя теста

Хорошее имя unit-теста отвечает на три вопроса:

1. **Что тестируется?**
2. **В каких условиях?**
3. **Какой ожидаемый результат?**

Один из самых распространённых и удачных шаблонов:

```
MethodName_StateUnderTest_ExpectedBehavior
```

Пример из TodoList:

```
[Fact]
public void AddTask_EmptyTitle_ThrowsArgumentException()
```

Даже без тела теста понятно:

- тестируется `AddTask`;
- входное состояние — пустое название;
- ожидаемое поведение — выбрасывается исключение.

Именование параметризованных тестов

Параметризованные тесты описывают **общее поведение**, а не конкретный пример.
Поэтому их имя должно быть обобщённым.

Плохой пример:

```
[Theory]
public void AddTask_InvalidTitle1()
```

Хороший пример:

```
[Theory]
public void AddTask_InvalidTitle_ThrowsArgumentException()
```

Конкретные значения уже отражены в `InlineData`, имя теста описывает **смысл**.

Имена тестов и уровень абстракции

Имя теста не должно:

- повторять внутреннюю реализацию;
- ссылаться на приватные методы;
- описывать технические детали.

Плохой пример:

```
public void AddTask_UsesListAdd()
```

Хороший пример:

```
public void AddTask_ValidTitle_TaskIsAdded()
```

Тест описывает поведение, а не способ его реализации.

Практическое задание

Что нужно сделать

1. Подготовка проекта
    1. Откройте решение с вашим проектом `TodoList`.
    2. Добавьте новый тестовый проект xUnit.
    3. Назовите его `TodoList.Tests`.
    4. Добавьте ссылку на основной проект.
    5. Убедитесь, что тесты обнаруживаются в Обозревателе Тестов.
2. Напишите unit-тесты для методов следующих классов:

    - `Profile`
    - `TodoItem`
    - `TodoList`
    - `CommandParser`

    Нельзя тестировать:

    - `Console.ReadLine`
    - `Console.WriteLine`
    - реальные файлы
    - реальную базу данных
3. Структура тестового проета
   Для каждого тестируемого класса в основном проекте, создайте соответствующий тестовый класс в тестовом проекте.

4. Используйте паттерн AAA для написания тестов
   Каждый тест должен быть разделен на:

```
// Arrange
// Act
// Assert
```

5. Именование тестов
   Имя теста должно отвечать на три вопроса:

```
MethodName_Scenario_ExpectedResult
```

```
Пример:
```

```
AddTask_WithEmptyTitle_ThrowsException
```

6. Параметризованные тесты
   Используйте параметризованные тесты для тестирования методов парсинга команд