

Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «Кубанский государственный технологический университет»

Кафедра информационных систем и программирования

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания по выполнению лабораторных работ
для студентов всех форм обучения
направлений 09.03.03 Прикладная информатика и
09.03.04 Программная инженерия

Краснодар
2023

Составитель: ст. преп. А. Г. Волик

Тестирование и отладка программного обеспечения:
методические указания по выполнению лабораторных работ для студентов
всех форм обучения направлений 09.03.03 Прикладная информатика и
09.03.04 Программная инженерия / Сост.: А. Г. Волик; Кубан. гос. технол.
ун-т. Каф. информационных систем и программирования. – Краснодар:
Изд. КубГТУ, 2023 – 248 с.

Методические указания составлены в соответствии с рабочей
программой дисциплины «Тестирование и отладка программного
обеспечения» для студентов направлений 09.03.03 Прикладная
информатика и 09.03.04 Программная инженерия.

Содержат краткие описания лабораторных работ, указания к их
выполнению, задания и требования к оформлению отчета. Рассмотрены
средства отладки в интегрированных средах разработки, подходы к
тестированию программного обеспечения, методы составления наборов
тестовых данных, которые обеспечивают достаточное покрытие кода,
индустриальный подход к тестированию, а также подходы и средства для
автоматизации процесса тестирования.

Ил. 54. Табл. 12. Библиогр.: 7 назв. Прил. 3

Печатается по решению методического совета ФГБОУ ВО
«Кубанский государственный технологический университет»

Рецензенты: д-р техн. наук, проф.,
профессор кафедры ИСП КубГТУ В. Н. Марков;
руководитель отдела телекоммуникаций Краснодарского
регионального центра сети «Консультант-Плюс»,
канд. техн. наук Н. Ф. Григорьев

© Волик А.Г., 2012-2023
© КубГТУ, 2023

Содержание

Введение.....	4
Лабораторная работа № 1. Отладка программ в интегрированных средах разработки.....	6
Лабораторная работа № 2. Программная отладка и трассировка программ	24
Лабораторная работа № 3. Тестирование методом черного ящика	37
Лабораторная работа № 4. Тестирование методом белого ящика	52
Лабораторная работа № 5. Тестирование, управляемое данными (Data-Driven Unit Tests) и Анализ покрытия кода (Code Coverage)	66
Лабораторная работа № 6. Тестирование методом серого ящика	86
Лабораторная работа № 7. Модульное тестирование объектно-ориентированных программ.....	88
Лабораторная работа № 8. Тестирование с использованием тестовых двойников (Test Doubles)	97
Лабораторная работа № 9. Автоматизация создания тестовых двойников (Test Doubles).....	128
Лабораторная работа № 10. Упрощение создания тестов при помощи библиотеки Fluent Assertions.....	137
Лабораторная работа № 11. Мутационное тестирование	148
Лабораторная работа № 12. Тестирование программ при отсутствии исходного кода методом черного ящика	162
Приложение А (справочное) Работа с системой управления репозиториями кода gitlab	174
Приложение Б (справочное) Автоматизация процесса тестирования	191
Приложение С (справочное) Работа с системой управления пакетами NuGet	236

Введение

Методические указания содержат описание лабораторных работ по дисциплине «Тестирование и отладка программного обеспечения» для направлений 09.03.03 Прикладная информатика и 09.03.04 Программная инженерия.

Целью выполнения лабораторных работ является закрепление основ и углубление знаний в области тестирования программного обеспечения, а так же ознакомление студентов со средствами автоматизации процесса тестирования и индустриального подхода к разработке и тестированию программ. В лабораторных работах внимание уделяется составлению наборов тестовых данных, достаточных для обеспечения надлежащего качества программного продукта, а также рассматривается подход к разработке программ, основанный на первенстве написания тестов перед написанием кода (TTD – Test Driven Development).

Целью тестирования является, в отличие от общепринятого мнения, не доказательство корректности работы программы, а, наоборот, поиск и локализация ошибок. Писать программы, которые покрыты достаточным количеством тестов без использования автоматизации сложно, поэтому большое внимание уделяется автоматизации процесса тестирования. Помимо этого в настоящее время усиливается тенденция к более внимательному отношению к качеству тестирования программ. Поэтому дисциплина «Тестирование и отладка программного обеспечения» является важным этапом подготовки студентов направлений 09.03.03 Прикладная информатика и 09.03.04 Программная инженерия.

При выполнении лабораторных работ должен соблюдаться следующий порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить номер варианта задания у преподавателя;
- изучить необходимый теоретический материал, пользуясь настоящими указаниями и рекомендованной литературой;
- написать программу и отладить ее на ЭВМ;
- подготовиться к ответам на теоретические вопросы по теме лабораторной работы;
- оформить отчет.

Все студенты должны предъявить индивидуальный отчет о результатах выполнения лабораторной работы. Допускается предъявление отчета в виде электронного документа.

Отчет должен содержать следующие пункты:

- 1) Титульный лист.

- 2) Наименование и цель работы.
- 3) Краткое теоретическое описание.
- 4) Задание на лабораторную работу, включающее четкую формулировку задачи.
- 5) Результаты выполнения работы.
- 6) Результаты выполнения тестирования программы.
- 7) Листинг программы.
- 8) Листинг модульных тестов.

При сдаче отчета студент должен показать знание теоретического материала в объеме, определяемом тематикой лабораторной работы, а также пониманием сущности выполняемой работы.

Лабораторная работа № 1. Отладка программ в интегрированных средах разработки

1 Цель работы

Цель работы – изучить инструментальные средства и возможности отладки программ в интегрированной среде Microsoft Visual Studio или JetBrains Rider.

2 Краткая теория

Интегрированные интерактивные среды разработки программ (IDE) включают в себя ряд средств, облегчающих процесс нахождения ошибок в программе, которые не позволяют ей корректно работать.

Как Microsoft Visual Studio, так и JetBrains Rider имеют в своем составе схожий набор средств отладки.

2.1 Понятие отладки

Отладка – это процесс поиска и исправления ошибок в программе, препятствующих корректной работе программы.

Отладка программы является одним из наиболее важных и трудоемких этапов разработки. Трудоемкость и эффективность отладки напрямую зависит от способа отладки и от средств языка программирования.

Существует ряд простых вещей, которые могут облегчить отладку Вашей программы. Для большинства случаев справедливо то, что не следует располагать на одной строке программы более одного оператора. Так как выполнение программы в процессе отладки происходит строка за строкой, это требование будет обеспечивать выполнение не более одного оператора каждый раз.

В то же время допускаются и случаи, когда можно разместить на строке несколько операторов. Если есть список операторов, которые нужно выполнить, но которые фактически не имеют отношения к отладке, то Вы можете произвольно располагать их на одной или двух строках так, чтобы их можно было быстрее пройти при пошаговом выполнении.

В конце концов, лучшей отладкой является профилактическая отладка. Хорошо разработанная, ясно написанная программа может иметь не только немного ошибок, но и будет облегчать для Вас трассировку и фиксацию местоположения этих ошибок. Существует несколько основных положений, о которых следует помнить при составлении программы:

- программируйте с постепенным наращиванием. При возможности кодируйте и отлаживайте программу небольшими

секциями. Прорабатывайте каждую секцию до конца, прежде чем переходить к следующей;

- разбивайте программу на части: модули, классы, методы, процедуры, функции. Избегайте построения функций, размер которых больше 25-30 строк, в противном случае разбивайте их на несколько меньших по размеру функций;
- старайтесь передавать информацию только через параметры, вместо использования глобальных переменных внутри процедур и функций. Это поможет Вам избежать побочных явлений и облегчит отладку программы, так как Вы сможете легко прослеживать всю информацию, входящую и выходящую из заданной процедуры или функции;
- не торопитесь. Сосредоточьте действия на том, чтобы программа работала правильно, прежде чем предпринимать шаги по ускорению ее работы.

2.2 Разновидности ошибок

Существует три основных типа ошибок: ошибки этапа компиляции, ошибки этапа выполнения и логические ошибки.

2.2.1 Ошибки этапа компиляции

Ошибки этапа компиляции или синтаксические ошибки происходят, когда исходный код нарушает правила синтаксиса языка. Компилятор не может скомпилировать программу, пока она не будет содержать допустимые операторы и иметь правильную структуру. Когда компилятор встречает оператор, который он не может распознать, то в окно Output среди разработки, заносится номер строки, в которой была найдена ошибка и описание ошибки. Дважды кликнув на запись об ошибке, вы попадаете на строку, в которой она произошла.

Наиболее общей причиной ошибок этапа компиляции являются ошибки набора (опечатки), пропущенные точки с запятой, ссылки на неописанные переменные, передача неверного числа (или типа) параметров функции и присваивание переменной значений неверного типа.

После устранения в программе всех синтаксических ошибок и ее успешной компиляции программа будет готова к выполнению и поиску ошибок этапа выполнения и логических ошибок.

2.2.2 Ошибки этапа выполнения

Ошибки этапа выполнения или семантические ошибки происходят, когда после компиляции полной программы, при ее выполнении делается что-то недопустимое. То есть, программа содержит допустимые операторы, но при выполнении операторов что-то происходит неверно. Например, программа может пытаться выполнить деление на ноль или открыть для ввода несуществующий файл.

Когда среда разработки обнаруживает такую ошибку, он завершает выполнение и выводит окно с сообщением о типе ошибки и указывает на место в программе, в которой она произошла (см. рисунки 1.1 и 1.2).

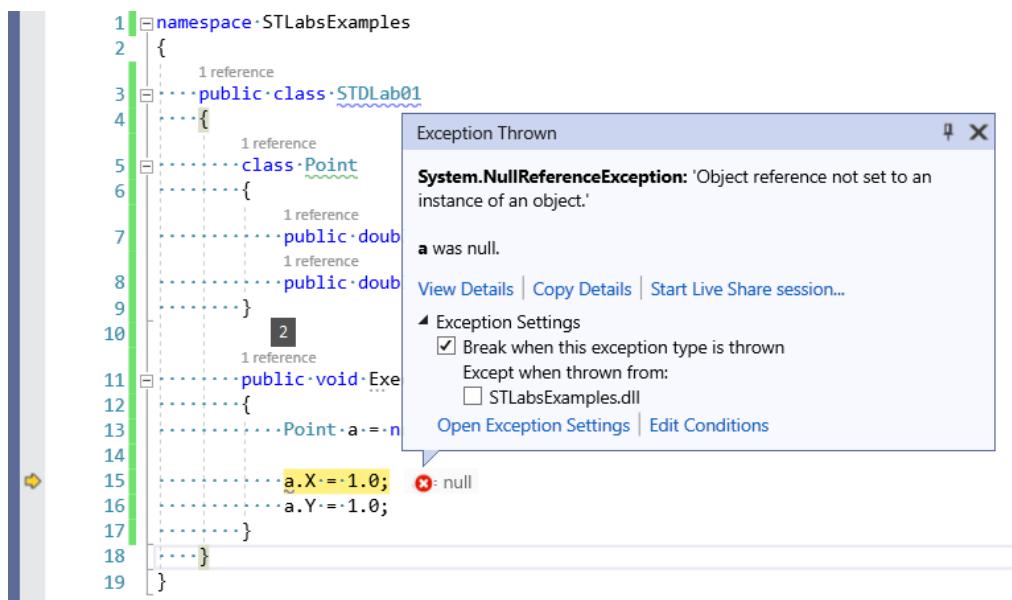


Рисунок 1.1 – Окно ошибки времени выполнения (Visual Studio 2019)

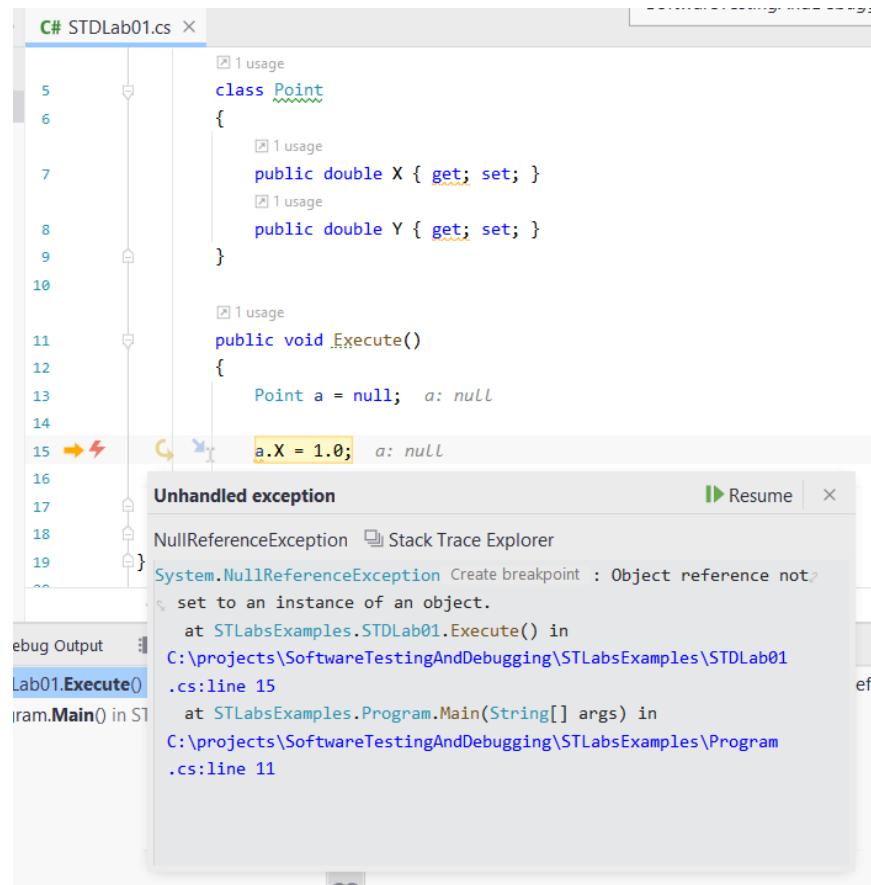


Рисунок 1.2 – Окно ошибки времени выполнения (Rider 2021)

2.2.3 Логические ошибки

Логические ошибки – это ошибки проектирования и реализации программы. То есть, все операторы допустимы и что-то делают, но не то, что предполагалось. Эти ошибки часто трудно отследить, поскольку IDE не может найти их автоматически, как синтаксические (ошибки в написании идентификаторов) и семантические (ошибки в написании конструкций языка) ошибки. К счастью, IDE включает в себя средства отладки, помогающие найти логические ошибки.

Логические ошибки приводят к некорректному или непредвиденному значению переменных, неправильному виду графических изображений или невыполнению кода, когда это ожидается. Далее рассматриваются методы отслеживания этих логических ошибок.

2.3 Методы отладки

Иногда, когда программа делает что-то непредвиденное, причина достаточно очевидна, и можно быстро исправить код программы. Но другие ошибки более трудноуловимы и вызываются взаимодействие различных частей программы. В этих случаях лучше всего остановить

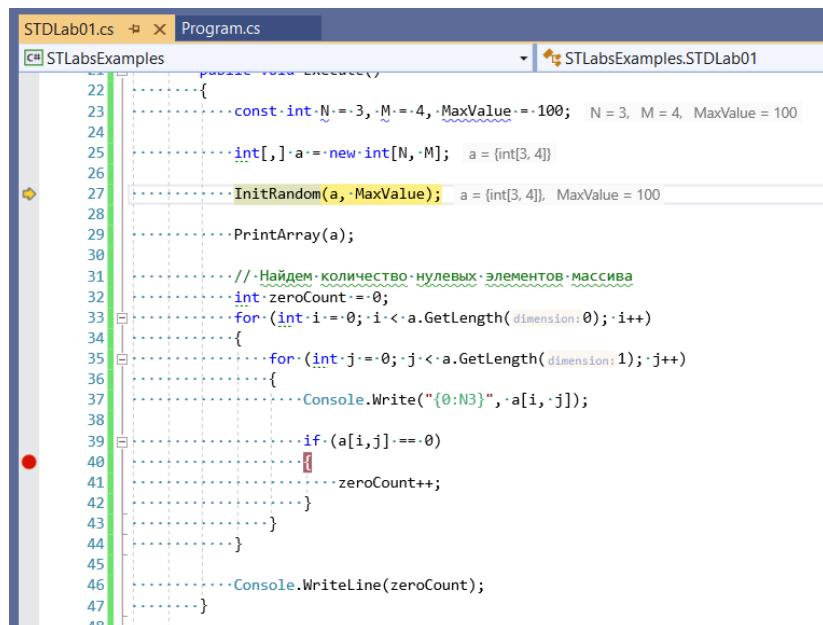
программу в заданной точке, пройти ее шаг за шагом и просмотреть состояние переменных и выражений. Такое управляемое выполнение – ключевой элемент отладки.

2.3.1 Установка точки прерывания

Точка прерывания позволяет остановить выполнение программы перед любой выполняемой инструкцией (оператором) с тем, чтобы продолжать выполнение программы либо в пошаговом режиме, либо в непрерывном режиме до следующей точки прерывания.

Чтобы задать точку прерывания перед некоторым оператором, необходимо установить перед ним текстовый курсор и нажать клавишу F9 или кликнуть мышью на левом боковом поле окна редактирования напротив оператора. Точка прерывания обозначается в виде красного кружка на левом поле окна редактирования, а также красным фоном выделяется оператор, при выполнении которого сработает точка остановки (см. рисунки 1.3 и 1.4). Повторное действие (щелчок на указанной кнопке или нажатие F9) снимает точку прерывания. В программе может быть несколько точек прерывания.

Для просмотра всех точек остановки необходимо выбрать пункт Debug | Windows | Breakpoints или нажать комбинацию клавиш Ctrl+Alt+B.



The screenshot shows the Visual Studio 2019 code editor with the file 'Program.cs' open. A red circular breakpoint icon is visible on the left margin next to line 40. The code is as follows:

```
22     public void execute()
23     {
24         const int N = 3, M = 4, MaxValue = 100; N = 3, M = 4, MaxValue = 100
25         int[,] a = new int[N, M]; a = {int[3, 4]}
26
27         InitRandom(a, MaxValue); a = {int[3, 4]}, MaxValue = 100
28
29         PrintArray(a);
30
31         // Найдем количество нулевых элементов массива
32         int zeroCount = 0;
33         for (int i = 0; i < a.GetLength(dimension: 0); i++)
34         {
35             for (int j = 0; j < a.GetLength(dimension: 1); j++)
36             {
37                 Console.Write("{0:N3}", a[i, j]);
38
39                 if (a[i, j] == 0)
40                     zeroCount++;
41             }
42         }
43     }
44
45     Console.WriteLine(zeroCount);
46
47 }
```

Рисунок 1.3 – Окно редактора с установленной точкой прерывания (Visual Studio 2019)

```
C# STDLab01.cs X
23
24
25
26
27 ➡  ↴ ↵ ↳
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
const int N = 3, M = 4, MaxValue = 100;
int[,] a = new int[N, M]; a: int[3, 4]
InitRandom(a, MaxValue); a: int[3, 4]
PrintArray(a);

// Найдем количество нулевых элементов массива
int zeroCount = 0; zeroCount: 0
for (int i = 0; i < a.GetLength(dimension: 0); i++)
{
    for (int j = 0; j < a.GetLength(dimension: 1); j++)
    {
        Console.WriteLine("{0:N3}", a[i, j]);
        if (a[i, j] == 0)
        {
            zeroCount++;
        }
    }
}
Console.WriteLine(zeroCount);
```

Рисунок 1.4 – Окно редактора с установленной точкой прерывания (Rider 2021)

2.3.2 Выполнение программы до точки прерывания

Программа запускается в отладочном режиме с помощью команды Debug | Start Debugging (или нажатием клавиши F5).

В результате код программы выполняется до строки, на которой установлена точка прерывания. Затем программа останавливается и отображает в окне Editor ту часть кода, где находится точка прерывания, причем желтая стрелка на левом поле указывает на строку (см. рисунок 1.2), которая будет выполняться на следующем шаге отладки.

2.3.3 Прекращение отладки

В ходе сеанса отладки иногда желательно начать все сначала. Выбор команды Debug | Stop Debugging или нажатие клавиши Shift+F5 приведет к полному сбросу, так что выполнение по шагам, или трассировка прекратится.

2.3.4 Пошаговое выполнение программы и трассировка

Команды выполнения по шагам Step Over (клавиша F10) и трассировки Trace Into (клавиша F11) меню Debug дают возможность построчного выполнения программы. Единственное отличие выполнения по шагам и трассировки состоит в том, как они работают с вызовами функций.

Выполнение по шагам вызова функции интерпретирует вызов как простой оператор и после завершения подпрограммы возвращает управление на следующую строку. Трассировка подпрограммы загружает код этой подпрограммы и продолжает ее построчное выполнение.

Нажимая клавишу F10, можно выполнять один оператор программы за другим. Предположим, что при пошаговом выполнении программы вы дошли до строки, в которой вызывается некоторая функция `func1()`. Если вы хотите пройти через код вызываемой функции, то надо нажать клавишу F11, а если внутренняя работа функции вас не интересует, а интересен только результат ее выполнения, то надо нажать клавишу F10.

Допустим, что вы вошли в код функции `func1()`, нажав клавишу F11, но через несколько строк решили выйти из него, т.е. продолжить отладку после возврата из функции. В этом случае надо нажать клавиши Shift+F11 или выбрать пункт Debug | Step Out.

2.3.5 Выполнение программы до курсора

Иногда, конечно, нежелательно выполнять по шагам всю программу только для того, чтобы добраться до того места, где возникает проблема. Отладчик позволяет выполнить сразу большой фрагмент программы до той точки, где необходимо начать выполнение по шагам.

Существует возможность пропустить пошаговое выполнение некоторого куска программы: установите текстовый курсор в нужное место программы и нажмите клавиши Ctrl+F10. Программа будет выполнена до курсора и остановится в данной точке ожидая дальнейших действий пользователя.

Причем, это можно сделать как в начале сеанса отладки, так и когда уже часть программы выполнена по шагам или протрассирована.

Чтобы продолжить дальнейшее выполнение программы без пошагового режима нажмите F5.

2.3.6 Отслеживание значений переменных во время выполнения программы

Выполнение программы по шагам или ее трассировка могут помочь найти ошибки в алгоритме программы, но обычно желательно также знать, что происходит на каждом шаге со значениями отдельных переменных.

Для наблюдения за выводом программы встроенный отладчик имеет средства для просмотра значений переменных, выражений и структур данных.

Чтобы узнать значение переменной задержите над ней указатель мыши. Рядом с именем переменной на экране появляется подсказка со значением этой переменной.

Часто программисту необходимо отслеживать значение конкретной переменной или выражения при выполнении программы по шагам.

Помимо экранной подсказки, переменные со своим значением отображается в окне Locals, расположенному в левом нижнем углу экрана Visual Studio (см. рисунок 1.5). В JetBrains Rider аналогичное окно располагается в правом нижнем углу (см. рисунок 1.6). В этом окне приведены значения последних переменных, с которыми работал Visual C#. Кроме этого, в окне «Watch 1» (которое находится также в левом нижнем углу, но на другой вкладке) можно задать имя любой переменной, за значениями которой вы хотите понаблюдать.

В Visual Studio также можно добавить переменную в список просмотра «Watch 1» нажав правую кнопку мыши над именем переменной и выбрав пункт «Add Watch». Кроме этого в окне просмотра «Watch 1» можно добавлять или удалять отслеживаемые элементы. В этом диалоговом окне можно проверять переменные и выражения и изменять значения любых переменных, включая строки, указатели, элементы массива и поля записей, что позволяет проверить реакцию программы на различные условия.

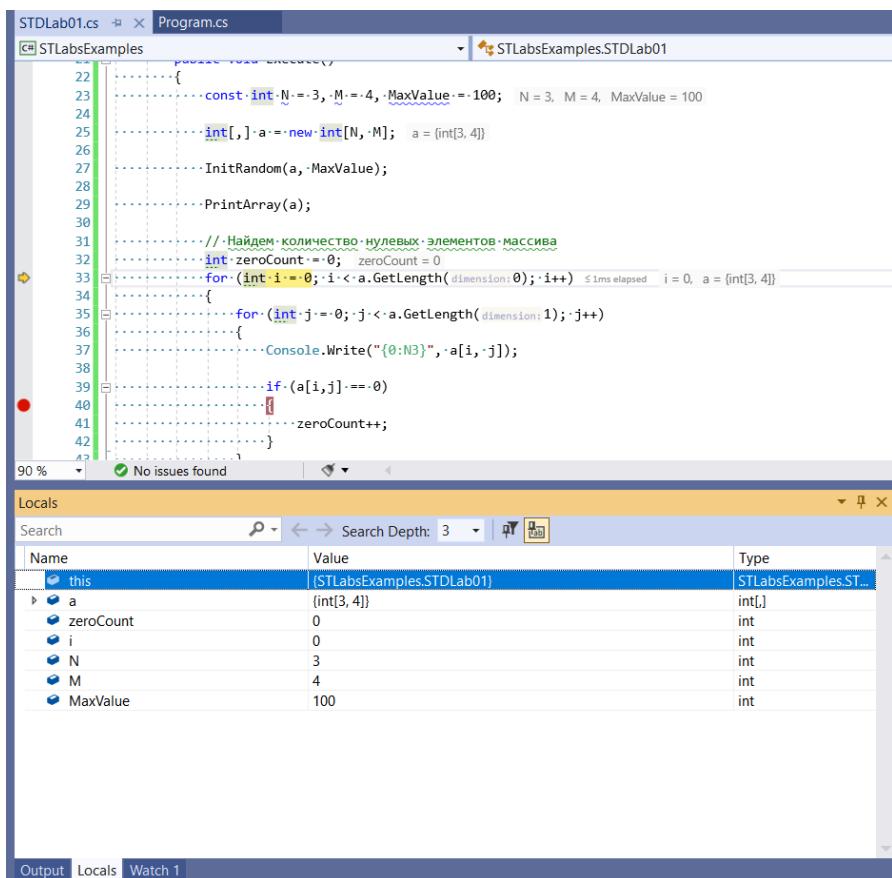


Рисунок 1.5 – Окно редактора и список отслеживаемых переменных (Visual Studio 2019)

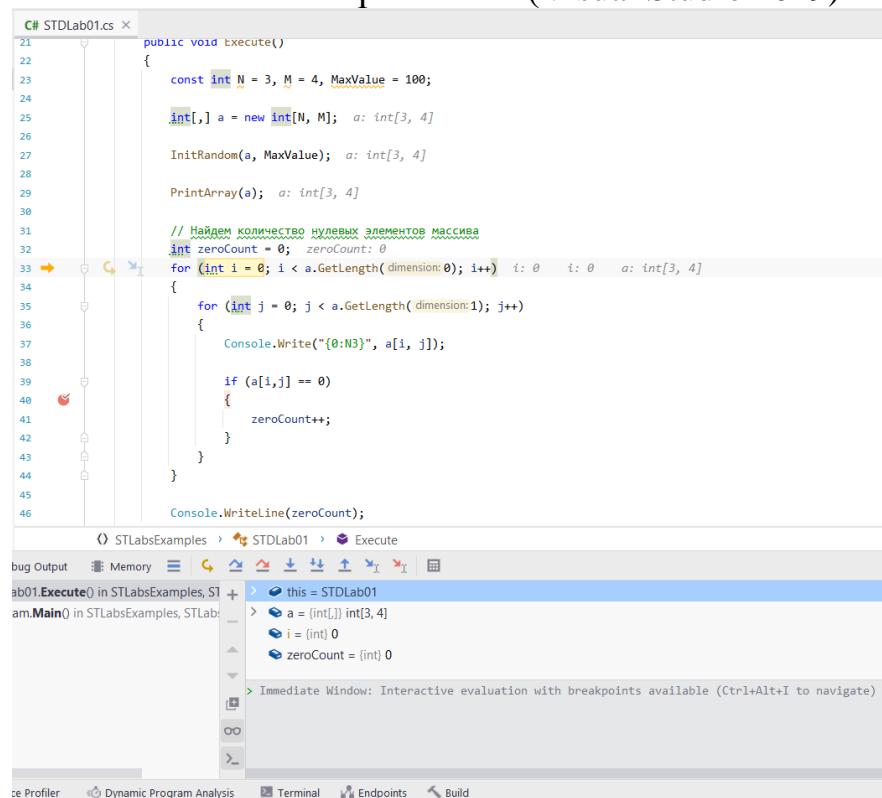


Рисунок 1.6 – Окно редактора и список отслеживаемых переменных (Rider 2021)

Оба средства вычисление и просмотра работают на уровне выражений, поэтому важно определить, что считается выражением. Выражение состоит из констант, переменных и структур данных, скомбинированных с помощью операций и большинства встроенных функций. Почти все, что можно использовать в правой части оператора присваивания, может также использоваться в качестве отладочного выражения.

Во время отладки можно изменить значение переменной. Для этого нужно ввести новое значение переменной в столбце Value, после чего оно отобразится красным цветом.

2.3.7 Создание условной точки останова

Среда разработки поддерживает создание дополнительных условий для точек остановки. Задав точку остановки, вы можете указать дополнительное условие ее срабатывания. В Visual Studio, нажав правой кнопкой мыши над строкой с точкой остановки, в меню Breakpoint появятся новые подпункты (см. рисунок 1.7). В этих точках останова можно задать несколько дополнительных параметров:

Location – место остановки (строка и модуль для остановки);

Actions – действия при срабатывании точки остановки);

Conditions – проверка условий изменения. Имеет выпадающее меню с набором опций (см. рисунок 1.8):

- Conditional Expression – проверка условие изменения (на равенство какому-либо значению или на изменение значения);

- Hit Count – проверка на счетчик проходов;

- Filter – комбинация нескольких условий для значений переменной.

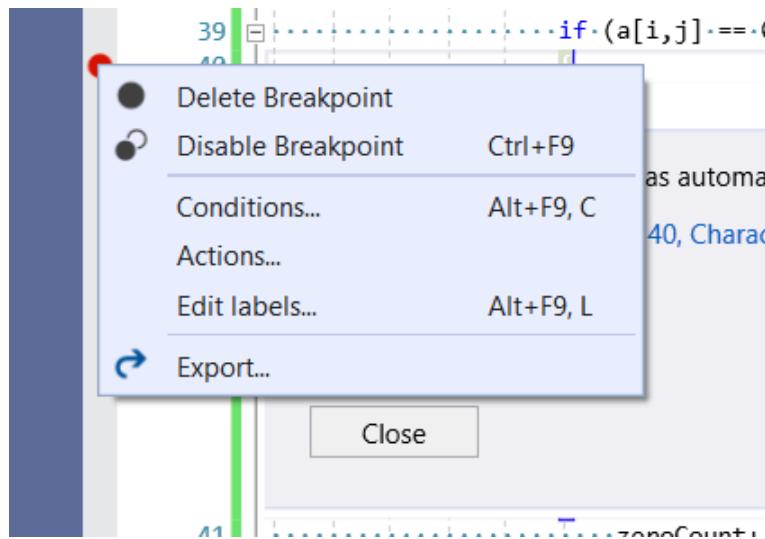


Рисунок 1.7 – Добавление дополнительных условий для точек остановки (Visual Studio 2019)



Рисунок 1.8 – Дополнительные условия для точек остановки (Visual Studio 2019)

В Rider имеются аналогичные возможности (см. рисунок 1.9).

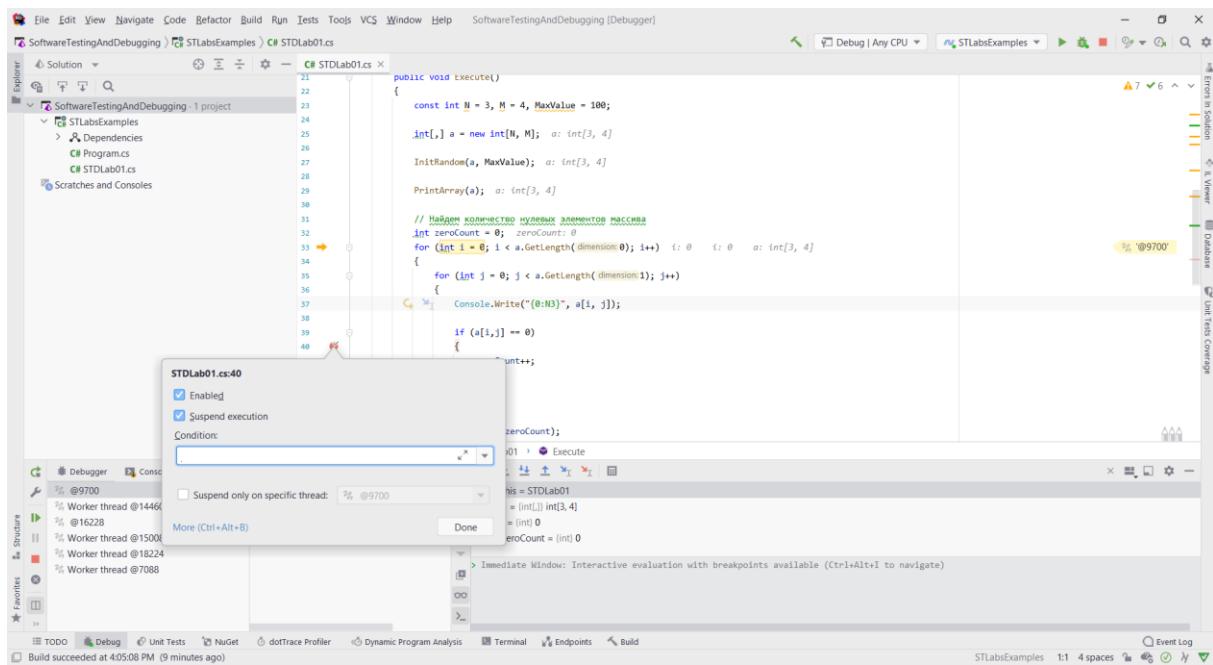


Рисунок 1.9 – Дополнительные условия для точек остановки (Rider 2021)

2.4 Работа с отладчиком

При отладке программы, наименьшим выполняемым элементом является оператор. Поэтому, если на одной строке программы содержится несколько операторов, можно отладить эти операторы индивидуально. Но для удобства, можно разбить оператор на несколько строк, которые будут выполняться пошагово.

2.4.1 Выполнение программы по шагам без захода в функцию

Это простейший способ выполнения программы по элементарным фрагментам. Выбор команды Debug | Step Over или нажатие клавиши F10 вызывает выполнение отладчиком всего кода в операторе, указанном строкой выполнения, включая любые вызываемые на ней процедуры или функции, пока управление не вернется обратно к программисту. После этого строка выполнения указывает **следующий** выполняемый оператор.

Рассмотрим следующий пример программы.

Пример 1.1. Простая программа, выполняемая по шагам

```
using System;
using System.Text;

namespace SoftwareTestingLabsExamples01x01
{
    class Program
    {
```

```
static int sqr(int x) //10
{
    int q = x * x; //11
    return q; //12
} //13

static void Main(string[] args) //0
{
    const int N = 10; //1
    int[] a = { 5, 2, 7, -9, 4, 8, -1, 0, 3, 6 }; //2
    //Найдем сумму квадратов //3
    //положительных элементов массива //4
    int s = 0; //5
    for (int i = 0; i < N; i++) //6
        if (a[i] > 0) s += sqr(a[i]); //7
    Console.WriteLine("Сумма квадратов равна: {0}", s); //8
}
} //9
```

Если нажать клавишу F10, то строка выполнения перемещается на фигурную скобку в начале программы (строка 0), поскольку это первое, что выполняется в программе. Второе нажатие клавиши F10 перемещает строку выполнения вниз до оператора объявления константы размерности массива N на следующей строке (строка 1). После этого нажатие F10 переводит указатель к строке с объявлением массива и его инициализации (строка 2). Далее, при нажатии F10, строки 3 и 4 будут пропущены, так как они состоят только из комментариев и указатель переместится к строке 5. В ней производится объявление переменной для хранения суммы квадратов элементов массива S и ее инициализация нулем.

После этого нажатие F10 вызывает начало выполнения цикла `for`. Первое нажатие инициализирует переменную `i`, и указатель переходит к строке 7. Далее нажатие F10 приводит к выполнению оператора `if` и указатель переходит на строку 8. Однако далее управление будет передано обратно на строку 6, так как цикл еще не завершен.

Обратите внимание, что в окне Locals выводятся значения счетчика *i*, максимального элемента *m*, размерности массива *N* и сам массив *a*.

Сравнение (строка 7) вызывается 10 раз, но невозможно понять, выполняется ли при этом операция присвоение нового значения m . Выполнение по шагам не позволяет отладчику показывать детали любых вычислений для отдельной строки.

Выполнение по шагам вызывает выполнение всего оператора сразу, поэтому невозможно видеть изменения в ходе выполнения цикла.

Если необходимо видеть подробности, то в пример нужно внести следующее простое изменение:

Пример 1.2. Модифицированная программа, для более удобного выполнения по шагам

```
using System;
using System.Text;

namespace SoftwareTestingLabsExamples01x02
{
    class Program
    {
        static int sqr(int x)
        {
            int q = x * x; //10
            return q; //11
        } //13

        static void Main(string[] args)
        {
            const int N = 10; //0
            int[] a = { 5, 2, 7, -9, 4, 8, -1, 0, 3, 6 }; //1
            //Найдем сумму квадратов //2
            //положительных элементов массива //3
            int s = 0; //4
            for (int i = 0; i < N; i++) //5
                if (a[i] > 0) //7.0
                    s += sqr(a[i]); //7.1
            Console.WriteLine("Сумма квадратов равна: {0}", s); //8
        } //9
    }
}
```

Если теперь нажимать клавишу F10, то указатель после строки 7.0 будет при выполнении условия переходить на строку 7.1, а при невыполнении на строку 8.

2.4.2 Выполнение программы по шагам с заходом в функцию (трассировка)

Чтобы выполнить трассировку кода, необходимо выбрать команду Debug | Trace Into или нажать клавишу F11.

Первое нажатие на F11 также передает управление на фигурную скобку основной программы (строка 0). Повторные нажатия F11 снова перемещают строку управления на те же операторы, что и F10, пока мы не доходим до строки 7.1. После этого нажатие клавиши F11 трассирует вызов функции `sqr` – строка выполнения перемещается на фигурную скобку в блоке функции (строка 10). Если продолжать нажимать F11, строка выполнения перемещается по функции, а затем, когда дойдет до

оператора `return` (при этом указатель будет располагаться напротив закрывающей фигурной скобки), возвращается к оператору вызова.

Следует обратить внимание на то, что трассировке все методы классов. Во избежание данного поведения необходимо использовать пошаговое выполнение (Step Over) по клавише F10, либо нажимать Shift + F11 для выхода из трассируемой функции.

3 Контрольные вопросы

- 1) Что такое отладка?
- 2) Для чего предназначена отладка?
- 3) Какие разновидности ошибок существуют?
- 4) Какие средства отладки предоставляет среда разработки Microsoft Visual Studio?
- 5) Какие средства отладки предоставляет среда разработки JetBrains Rider?
- 6) Что такое точка остановки (breakpoint)?
- 7) Какие дополнительные условия можно устанавливать в точке остановки?
- 8) Какие возможности существуют для слежения за значениями переменных во время отладки?
- 9) Как изменить значение переменной?
- 10) Какие горячие клавиши для работы с отладчиком вы знаете?

4 Задание

- 1) Выполнить задание в соответствии с пунктом 5.1.
- 2) Выполнить задание в соответствии с пунктом 5.2.
- 3) Оформить отчёт.

5 Задание для выполнения работы

В отчете должен присутствовать текст программы с номерами строк. Последовательность выполняемых строк записывается в столбец, с указанием значений изменяющихся переменных через запятую. При выполнении программы до точки остановки, в последовательности выполнения указать строку с точкой остановки и значения переменных в данной точке.

Например:

```
...  
const int N = 10; //1  
int a[N] = {5, 2, 7, -9, 4, 8, -1, 0, 3, 6}; //2  
...
```

Последовательность выполнения:

```
...
2: a = {5, 2, 7, 9, 4, 8, -1, 0, 3, 6}
5: i = 0
6: i = 0, s = 25
...
```

5.1 Часть первая

- 1) Набрать программу примера 1 (раздел 2.4.1)
- 2) Выполнить программу по шагам, фиксируя в отчете, в строке с каким номером находится строка выполнения при каждом нажатии на F10.
- 3) Внести в программу изменения в соответствии с примером 2.
- 4) Выполнить задание пункта 2.
- 5) Выполнить трассировку исходной программы примера 3 (раздел 2.4.2), фиксируя в отчете, в строке с каким номером находится строка выполнения при каждом нажатии на F11.

5.2 Часть вторая

- 1) Набрать и откомпилировать следующую программу:

```
using System;
using System.Text;

namespace SoftwareTestingLabsExamples01x03
{
    class Program
    {
        //Метод, считающий сумму элементов массива
        static int sum(int[] x, int N)
        {
            int s = 0;
            for (int i = 0; i < N; i++)
                s += x[i];
            return s;
        }
        //Метод для ввода целых чисел с клавиатуры
        static int ReadInt(string prompt)
        {
            Console.Write(prompt);
            int x = int.Parse(Console.ReadLine());
            return x;
        }

        static void Main(string[] args)
        {
```

```

const int N = 10;
int[] a = new int[N] { 1, 3, -5, 0, 4, 6, -1, 9, 3, 2 };

//Найдем максимальный элемент массива
int m = a[0];
for (int i = 1; i < N; i++)
    if (m < a[i])
        m = a[i];

Console.WriteLine(m);

//Найдем сумму элементов массива
int s;
s = sum(a, N);

Console.WriteLine(s);

int z = s / m;
int k = 0;

for (int i = 0; i < N; i++)
    if (a[i] > z)
        k += a[i];
    else
        k -= a[i];

Console.WriteLine(k);

int x, y;

x = ReadInt("");
y = ReadInt("");

s = 0;
while ((x != 0) && (y != 0))
{
    x--;
    y--;
    s += x + y;
}

Console.WriteLine(s);
}
}
}

```

- 2) После каждой строки программы проставить номер. Например,
 //1, //2 и т.д.

- 3) Выполнить трассировку программы (без захода в функции стандартных библиотек), наблюдая за переменными в окне Locals.
- 4) Остановить отладку программы.
- 5) Установить точку остановки на операторе if ($a[i] > z$).
- 6) Выполнить программу до курсора на строке $s = \text{sum}(a, N)$;
- 7) Продолжить выполнить программы до точки остановки. Далее продолжать пошаговое выполнение до строки `Console.WriteLine(k)`.
- 8) Добавить в окно Watch 1 переменные x, y, s для наблюдения изменения их значений. Продолжать пошаговое выполнение.
- 9) Остановить отладку программы.
- 10) В операторе цикла while задать условную точку останова по числу проходов. Запустить программу для отладки.
- 11) Продолжать пошаговое выполнение до конца программы, наблюдая изменение значений x, y, s в окне Watch 1.
- 12) Записать полученные результаты.
- 13) Остановить отладку программы.
- 14) В операторе цикла while задать еще одну условную точку останова по логическому условию. Запустить программу для отладки
- 15) Продолжать пошаговое выполнение до конца программы, наблюдая изменение значений x, y, s в окне Watch 1.
- 16) Записать полученные результаты.

Лабораторная работа № 2. Программная отладка и трассировка программ

1 Цель работы

Изучить средства и возможности программной отладки и трассировки программ.

2 Краткая теория

Помимо встроенного в среду разработки отладчика, предоставляющие средства отладки в интерактивном режиме, существуют средства для организации программной отладки. К таким средствам относятся стандартные классы .NET, такие как **Debug** и **Trace**.

Данный подход удобен в тех случаях, когда структура кода излишне сложна (например, имеется множество вложенных циклов со сложными условиями завершения) или возникают проблемы с локализацией ошибок (например, из-за разброса потенциальных проблемных точек по всему коду проекта). В таких случаях отладочная печать и программная остановка приложения позволяют упростить нахождение причины и место возникновения ошибки. Вы можете сами выбирать наборы данных, которые необходимы в данной точке программы, тогда как интерактивный режим отображает все заранее выбранные значения переменных и величин.

При этом наличие в коде самих операторов управления отладкой позволяет сохранять все точки останова и прочую информацию между сессиями работы с проектом.

Этот способ отладки исторически появился первым, до появления интегрированных сред разработки, однако по сей день пользуется заслуженной популярностью у программистов.

При этом следует разделять два сходных понятия: трассировка и отладка. **Трассировка** кода представляет собой получение информационных сообщений о работе приложения во время выполнения. **Отладка** – это отслеживание и устранение ошибок программирования в приложении при разработке.

2.1 Класс Debug

Класс **Debug** находится в пространстве имен **System.Diagnostics** и предоставляет статический класс с набором методов и свойств, помогающих при отладке кода. Когда приложение будет готово к выпуску, можно скомпилировать его, не включая условный атрибут **Debug**, чтобы код отладки не включался компилятором в конечный исполняемый файл.

Результаты своей работы он выводит в окно Output (рисунок 2.1).

The screenshot shows the Visual Studio IDE interface. The top part is the code editor with the file 'STLabsExamples01x01.Program' open. The code is written in C# and calculates the sum of squares of positive elements in an array. The bottom part is the 'Output' window, which displays the debug messages printed by the program. The messages include loading assembly information, thread exits, and the results of the debug prints.

```
22
23     static void Main(string[] args)
24     {
25         const int N = 10;
26         int[] a = {-5, -2, -7, -9, -4, -8, -1, -6, -3, -6};
27
28         //Определяем границы отладочного вывода
29         Debug.Print("Начало отладки");
30
31         //Найдем сумму квадратов положительных элементов массива
32         int s = 0;
33         for (int i = 0; i < N; i++)
34         {
35             //Выводим все отрицательные значения в окно отладки
36             Debug.WriteLineIf(a[i] < 0,
37                 String.Format("!Отрицательное значение: {0}", a[i]));
38
39             if (a[i] > 0)
40             {
41                 s += sqr(a[i]);
42             }
43         }
44
45         Console.WriteLine("Сумма квадратов равна: {0}", s);
46
47         Debug.Print("Конец отладки");
48         //Выводим сообщение об ошибке в программе
49         //Debug.Fail("Ошибка в работе программы");
50     }
```

Output

```
'STLabsExamples01x01.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System\2.0.0.0__b77a5c561934e089\System.dll'
'STLabsExamples01x01.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.NET\assembly\GAC_32\SystemCore\2.0.0.0__b77a5c561934e089\SystemCore.dll'
The thread 'vshost.NotifyLoad' (0x3e9c) has exited with code 0 (0x0).
The thread '' (0x3da0) has exited with code 0 (0x0).
The thread 'vshost.LoadReference' (0x22c8) has exited with code 0 (0x0).
'STLabsExamples01x01.vshost.exe' (Managed (v4.0.30319)): Loaded 'D:\projects\vsp\Visual Studio 2010\Projects\So...
'STLabsExamples01x01.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System\2.0.0.0__b77a5c561934e089\System.dll'
Начало отладки
!Отрицательное значение: -9
!Отрицательное значение: -1
Step into: Stepping over non-user code 'System.Diagnostics.Debug.Print'
Конец отладки
```

Рисунок 2.1 – Окно просмотра отладочных сообщений Output

Свойства и методы класса Assert, помогающие при отладке кода, представлены в таблицах 2.1 и 2.2 соответственно.

Таблица 2.1 – Свойства класса Assert

Имя	Описание
AutoFlush	Получает или задает значение, определяющее необходимость вызова метода Flush для свойства Listeners после каждой записи.
IndentLevel	Возвращает или задает уровень отступа.
IndentSize	Возвращает или задает число пробелов в отступе.
Listeners	Получает коллекцию прослушивателей, отслеживающих данные отладки.

Таблица 2.2 – Методы класса Assert

Имя	Описание
Assert(Boolean)	Проверяет условие; если условие имеет значение false, выводит сообщение, отображающее стек вызовов.
Assert(Boolean, String)	Проверяет условие; если условие имеет значение false, выводит указанное сообщения и отображает окно сообщения со стеком вызовов.
Assert(Boolean, String, String)	Проверяет условие; если условие имеет значение false, выводит два указанных сообщения и отображает окно сообщения со стеком вызовов.
Assert(Boolean, String, String, Object())	Проверяет условие; если условие имеет значение false, выводит два сообщения (простое и отформатированное) и отображает окно сообщения со стеком вызовов.
Close()	Очищает выходной буфер, а затем вызывает метод Close на каждый Listeners.
Fail(String)	Выдает указанное сообщение об ошибке.
Fail(String, String)	Выдает простое и подробное сообщение об ошибке.
Flush()	Очищает выходной буфер и вызывает запись данных буфера в коллекцию Listeners.
Indent()	Увеличивает текущее значение свойства IndentLevel на единицу.
Print(String)	Записывает сообщение, заканчивающееся ограничителем строки, в прослушиватели трассировки в коллекции Listeners.
Print(String, Object())	Записывает отформатированную строку, заканчивающуюся ограничителем строки, в прослушиватели трассировки в коллекции Listeners.
Unindent()	Уменьшает текущее значение свойства IndentLevel на единицу.
Write(Object)	Записывает значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners.
Write(String)	Записывает сообщение в прослушиватели трассировки в коллекции Listeners.
Write(Object, String)	Записывает имя категории и значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners.
Write(String, String)	Записывает имя категории и сообщение в прослушиватели трассировки в коллекции Listeners.

Продолжение таблицы 10.2

WriteIf(Boolean, Object)	Записывает значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners, если условие имеет значение true.
WriteIf(Boolean, String)	Записывает сообщение в прослушиватели трассировки в коллекции Listeners, если условие имеет значение true.
WriteIf(Boolean, Object, String)	Записывает имя категории и значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners, если условие имеет значение true.
WriteIf(Boolean, String, String)	Записывает имя категории и сообщение в прослушиватели трассировки в коллекции Listeners, если условие истинно (true).
WriteLine(Object)	Записывает значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners.
WriteLine(String)	Записывает сообщение, заканчивающееся ограничителем строки, в прослушиватели трассировки в коллекции Listeners.
WriteLine(Object, String)	Записывает имя категории и значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners.
WriteLine(String, Object())	Записывает форматированное сообщение, заканчивающееся ограничителем строки, в прослушиватели трассировки в коллекции Listeners.
WriteLine(String, String)	Записывает имя категории и сообщение в прослушиватели трассировки в коллекции Listeners
WriteLineIf(Boolean, Object)	Записывает значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners, если условие имеет значение true.
WriteLineIf(Boolean, String)	Записывает сообщение в прослушиватели трассировки в коллекции Listeners, если условие имеет значение true.
WriteLineIf(Boolean, Object, String)	Записывает имя категории и значение метода ToString объекта в прослушиватели трассировки в коллекции Listeners, если условие имеет значение true.
WriteLineIf(Boolean, String, String)	Записывает имя категории и сообщение в прослушиватели трассировки в коллекции Listeners, если условие имеет значение true.

Пример 2.1. Работа с классом Debug

```

using System;
using System.Diagnostics;

namespace STLabsExamples2x01
{
    class Program
    {
        static int sqr(int x)
        {
            //Ожидаем, что число на входе должно быть положительным
        }
    }
}

```

```

//В противном случае останавливаемся с ошибкой
Debug.Assert(x > 0);

int q = x * x;

//Ожидаем, что число на выходе должно быть положительным
//Квадрат числа всегда больше 0
Debug.Assert(q > 0);
return q;
}

static void Main(string[] args)
{
    const int N = 10;
    int[] a = { 5, 2, 7, -9, 4, 8, -1, 0, 3, 6 };

    //Определяем границы отладочного вывода
    Debug.Print("Начало отладки");

    //Найдем сумму квадратов положительных элементов массива
    int s = 0;
    for (int i = 0; i < N; i++)
    {
        //Выводим все отрицательные значения в окно отладки
        Debug.WriteLineIf(a[i] < 0,
            String.Format("!Отрицательное значение: {0}", a[i]));

        if (a[i] > 0)
        {
            s += sqr(a[i]);
        }
    }

    Console.WriteLine("Сумма квадратов равна: {0}", s);

    Debug.Print("Конец отладки");
    //Выводим сообщение об ошибке в программе
    //Debug.Fail("Ошибка в работе программы");
}
}
}

```

Отладочный вывод примера:

```

Начало отладки
!Отрицательное значение: -9
!Отрицательное значение: -1
Конец отладки

```

2.2 Класс Trace

Класс Trace также находится в пространстве имен System.Diagnostics и предоставляет статический класс с набором методов и свойств, помогающих при трассировке во время выполнения программы.

Трассировка важна в тех случаях, когда, например, возникновение ошибки не получается воспроизвести на компьютере разработчика, а на компьютере пользователя она возникает регулярно. Проанализировав вывод трассировщика можно сделать вывод о том, что приводит к возникновению ошибки. Трассировка позволяет отслеживать состояние приложения с реально заданными параметрами и наборами данных. С помощью трассировки можно выделять неполадки и устранять их, не влияя на процесс выполнения.

С помощью методов класса **Trace** можно также отслеживать выполнение кода в установленном приложении. Если в коде поместить «Переключатели трассировки», можно управлять трассировкой и ее функциями. Это позволяет отслеживать состояние приложения в производственной среде. Это особенно важно для бизнес-приложений, использующих различные компоненты, выполняющиеся на нескольких компьютерах. Файл конфигурации дает возможность управлять использованием параметров после развертывания.

При разработке приложения, для которого предполагается использовать трассировку, в код приложения, как правило, включаются сообщения трассировки и отладки. Когда приложение готово к развертыванию, можно скомпилировать построение выпуска без включения условного атрибута **Debug**. Однако можно включить условный атрибут **Trace**, чтобы компилятор вставил в исполняемый файл код трассировки.

Существует три этапа трассировки кода:

- Инструментирование — добавление в приложение кода трассировки.
- Трассировка — код трассировки записывает данные в заданном конечном расположении.
- Анализ — оценка сведений трассировки для выявления и понимания проблем, имеющихся в приложении.

Во время разработки все методы вывода трассировки и отладки по умолчанию записывают сведения в окне вывода Visual Studio. В развертываемом приложении эти методы записывают сведения в указанное конечное расположение.

Выходные данные трассировки собираются объектами, которые называются *прослушивателями* (*Listeners*). Прослушиватели представляют собой объекты, получающие выходные данные трассировки и

записывающие их в устройство вывода (как правило, в окно, журнал событий или текстовый файл). В момент создания прослушиватель трассировки, как правило, добавляется к коллекции **Trace.Listeners**, что позволяет ему получать все выходные данные трассировки.

Сведения трассировки всегда записываются, по крайней мере, в целевой объект вывода **DefaultTraceListener** класса **Trace**. Если по каким-то причинам объект **DefaultTraceListener** был удален без добавления других прослушивателей в коллекцию **Listeners**, сообщения трассировки не будут получены.

Применив собственный прослушиватель, пользователь может получить необходимые ему результаты. Пользовательский прослушиватель трассировки может, например, отображать сообщения в окне сообщений или подключаться к базе данных для добавления сообщений в таблицу.

Класс предоставляет набор методов и свойств, используемых при трассировке выполняемого кода. Методы и их назначение похожи на методы класса **Debug** за некоторым исключением:

- Refresh – Обновляет данные конфигурации трассировки.
- TraceError – Перегружен. Записывает информацию об ошибке в прослушиватели трассировки в коллекции **Listeners**.
- TraceInformation – Перегружен. Записывает информационное сообщение в прослушиватели трассировки в коллекции **Listeners**.
- TraceWarning – Перегружен. Записывает предупреждающую информацию в прослушиватели трассировки в коллекции **Listeners**.

Пример 2.2. Работа с классом Trace

```
using System;
using System.Diagnostics;

namespace STLabsExamples2x02
{
    class Program
    {
        //Метод, считающий сумму элементов массива
        static int sum(int[] x, int N)
        {
            Trace.Indent();
            Trace.WriteLine("Вывод суммы на каждом шаге");
            int s = 0;
            for (int i = 0; i < N; i++)
            {
                Trace.Write(String.Format("{0} => ", s));
                s += x[i];
            }
        }
    }
}
```

```

        Trace.WriteLine(String.Format("{0}", s));
        Trace.Unindent();
        return s;
    }

    static void Main(string[] args)
    {
        //Устанавливаем свой прослушиватель
        Trace.Listeners.Add(
            new TextWriterTraceListener(Console.Out));
        //Просушиватель будет выводить данные в файл trace.txt
        //StreamWriter txt = new StreamWriter(
        //    new FileStream("trace.txt", FileMode.OpenOrCreate));
        //Trace.Listeners.Add(new TextWriterTraceListener(txt));
        Trace.AutoFlush = true;
        Trace.Indent();

        //Начало трассировки
        Trace.TraceInformation("Начало трассировки");
        Trace.WriteLine(String.Format("Дата : {0}", DateTime.Now));
        const int N = 10;
        int[] a = new int[N] { 1, 3, -5, 0, 4, 6, -1, 9, 3, 2 };

        //Найдем максимальный элемент массива
        int m = a[0];
        for (int i = 1; i < N; i++)
            if (m < a[i])
                m = a[i];

        Console.WriteLine(
            String.Format("Максимальный элемент массива: {0}", m));

        //Найдем сумму элементов массива
        int s;
        s = sum(a, N);

        Console.WriteLine(
            String.Format("Сумма элементов массива: {0}", s));

        Trace.TraceInformation("Конец трассировки");
        Trace.WriteLine(String.Format("Дата : {0}", DateTime.Now));
        Trace.Unindent();
    }
}
}

```

2.3 Вычисление определенного интеграла

Рассмотрим различные способы параллельной реализации численного определения значения интеграла по квадратурной формуле

средних прямоугольников с заданным разбиением области интегрирования.

Пусть ставится задача вычисления значения интеграла вида $\int_a^b f(x)dx$.

Рассмотрим $f(x) = \ln(\frac{1}{x})$, а в качестве пределов интегрирования $a = 0$, $b = 1$. Поскольку $\int \ln(1/x)dx = x(\ln(1/x) + 1) + C$, очевидно, что интеграл при данных пределах интегрирования является несобственным и имеет значение, равное 1. Так как рассматривается несобственный интеграл, численное определение его значения с наперед заданной точностью становится непростой вычислительной задачей.

Построим в области интегрирования равномерную сетку и воспользуемся формулой средних прямоугольников (рисунок 2.2). При данном подходе приближенное значение искомого интеграла определяется по формуле $\int_0^1 f(x)dx \approx \frac{1}{N} \sum_{k=0}^{N-1} f\left(\frac{x_{k+1} + x_k}{2}\right)$, что позволяет не вычислять значения подынтегральной функции в узлах сетки.

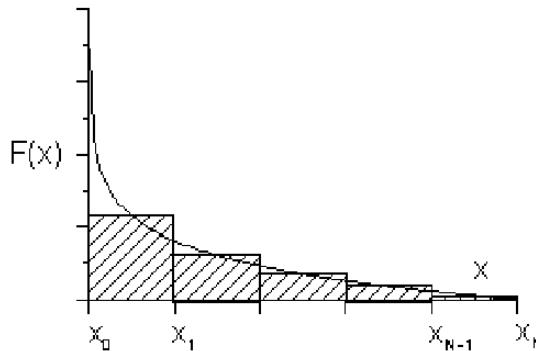


Рисунок 2.2 – Приближенное вычисление интеграла по формуле прямоугольников

Интеграл для метода прямоугольников вычисляется по формуле:

$$\int_a^b f(x)dx \approx \frac{(b-a)}{N} \sum_{k=0}^{N-1} f\left(\frac{x_{k+1} + x_k}{2}\right). \quad (2.1)$$

Интеграл для метода трапеций вычисляется по формуле:

$$\int_a^b f(x)dx \approx h \sum_{k=1}^N \frac{f(x_{k-1}) + f(x_k)}{2}, h = \frac{(b-a)}{N}. \quad (2.2)$$

Интеграл для метода Симпсона (парабол) вычисляется по формуле:

$$\int_a^b f(x)dx \approx \frac{h}{3} \left(f(x_0) + 4 \sum_{k=1}^{N/2} f(x_{2k-1}) + 2 \sum_{k=1}^{N/2-1} f(x_{2k}) + f(x_N) \right), h = \frac{(b-a)}{N}. \quad (2.3)$$

Обычно для равномерной сетки данную формулу записывают в других обозначениях, когда отрезок $[a, b]$, разбит на $2N$ узлов:

$$\int_a^b f(x)dx \approx \frac{h}{6} \left(f(x_0) + 4 \sum_{k=1}^N f(x_{2k-1}) + 2 \sum_{k=1}^{N-1} f(x_{2k}) + f(x_{2N}) \right), h = \frac{(b-a)}{N}. \quad (2.4)$$

2.4 Правило Рунге для оценки погрешности

Правило Рунге – правило оценки погрешности численных методов.

Основная идея состоит в вычислении приближения выбранным методом с шагом h , а затем с шагом $h/2$, и дальнейшем рассмотрении разностей погрешностей для этих двух вычислений.

Интеграл вычисляется по выбранной формуле при числе шагов, равном n , а затем при числе шагов, равном $2n$. Погрешность (при $2n$ шагов), определяется по формуле Рунге:

$$\Delta_{2n} \approx \Theta \cdot |I_{2n} - I_n|, \quad (2.5)$$

где $\Theta = \frac{1}{3}$ для формул прямоугольников и трапеций, $\Theta = \frac{1}{15}$ – для формулы Симпсона.

2.5 Арифметическое переполнение

Арифметическое переполнение – специфичная для компьютерной арифметики ситуация, когда при арифметическом действии результат становится больше максимально возможного значения для переменной, использующейся для хранения результата.

В качестве примера может служить сложение двух переменных размером 8 бит с записью результата в переменную того же размера. При сложении чисел, сумма которых превышает максимально допустимое значение (для 8-битного числа это 255) возникает переполнение.

8 7654 3210 – биты числа	
250	1111 1010
+ 32	0010 0000
-----	-----
282	1 0001 1010

Старший разряд не помещается в приемник, и отбрасывается. При этом в результате записывается не ожидаемое число 282 (1|0001|1010|), а 26 (0001|1010|). Если не проверять факт переполнения, то может быть

получен неверный результат вычислений или возникнуть логическая ошибка в программе.

Отсутствие проверки переполнения может привести к потенциальным ошибкам. Однако C# не выполняет автоматической проверки. Это обусловлено соображениями производительности, и данная проверка по умолчанию все же отключена (В противном случае проверка переполнения будет производиться при каждом преобразовании типов во всей программе, и даже там, где это не нужно).

Для включения проверки нужно в настройках проекта включить пункт "Check for arithmetic overflow/underflow" (Project properties => Build => Advanced).

Либо можно воспользоваться ключевым словом checked для операции вычисления:

```
int a = int.MaxValue;
int x = checked(a+b);
```

В этом случае при переполнении будет вызвано исключение System.OverflowException.

Однако в случае циклического вычисления суммы можно выполнять проверку самостоятельно, сравнивая полученный на текущем шаге результат с результатом (суммой) на предыдущем. В случае переполнения новая сумма будет меньше предыдущей (для простоты ограничимся 16-битными числами):

```
Int16 sum = 32767; // 0x7FFF
sum += 256;         // 0x80FF => ожидаемо 33 023, реально -32 513
```

Аналогично будет выглядеть переполнение для беззнаковых чисел:

```
UInt16 sum = 65535; // 0xFFFF
sum += 256;          // 0x(1)00FF Ожидаемо 65 791, реально 255
```

3 Контрольные вопросы

- 1) Чем отличается отладка от трассировки?
- 2) Для чего предназначена отладка?
- 3) Для чего предназначена трассировка?
- 4) Какие методы класса для отладки вы знаете?
- 5) Какие методы класса для трассировки вы знаете?
- 6) Для чего используется метод Debug.Assert?
- 7) Для чего используется метод Debug.Fail?
- 8) Для чего используется метод Trace.Error?
- 9) Для чего используется метод Trace.Warning?

10) Для чего используется метод Trace.Information?

4 Задание

4.1 Задание 1

Написать программу для приближенного вычисления значения интеграла $\int_a^b f(x)dx$ с заранее заданной точностью $\varepsilon > 0$, т.е. вычисления продолжаются до тех пор, пока $\Delta_{2n} > \varepsilon$, и прекращаются в тот момент, когда значение погрешности Δ_{2n} из формулы Рунге становится меньше либо равно ε . В лабораторной работе читать $\varepsilon = 0.0001$.

Функцию выбрать из вариантов в разделе 5 в соответствии с номером в журнале.

Для вычислений использовать формулу приближенного вычисления в соответствии с номером в журнале по модулю 3 (№%3):

- 0) Обобщенная формула прямоугольников.
- 1) Обобщенная формула Симпсона.
- 2) Обобщенная формула трапеций.

Оценить погрешность вычислений с помощью правила Рунге.

На каждой итерации производить трассировку всех значений изменяющихся переменных.

Вовремя работы алгоритма проверять выход за границы интегрирования (интервал $[a;b]$) переменной x при помощи Assert.

При вычислении значения погрешности и интеграла выводить информацию в лог трассировки на каждой итерации вычислений.

При возникновении ошибок также выводить их в лог трассировки.

Пусть FN – номер первой буквы имени, а LN – номер первой буквы фамилии студента. Найти значение интеграла на FN-ом шаге и вывести его в вывод отладчика. Найти значение интеграла на LN-ом шаге и вывести его в вывод трассировщика.

4.2 Задание 2

Написать программу для вычисления суммы элементов последовательности. Последовательность выбрать из вариантов в разделе 5.2 в соответствии с номером в журнале.

В программе проверить ситуацию арифметического переполнения при помощи Assert и при возникновении ошибки выводить в лог трассировки сообщение.

5 Варианты заданий

5.1 Численное интегрирование

- 1) $f(x) = e^{(-x^2+0.38)} / (2 + \sin(1/(1.5 + x^2))), a = 0.4, b = 1.0$
- 2) $f(x) = (x^2 + \sin(0.48(x + 2)) / e^{(x^2+0.38)}), a = 0.4, b = 0.6$
- 3) $f(x) = (1 - e^{(0.7/x)}) / (x + 2), a = 1.0, b = 3.0$
- 4) $f(x) = e^{-tg(0.84x)} / (0.35 + \cos(x)), a = 0.0, b = \pi/2$
- 5) $f(x) = arctg(0.7x) / (x + 1.48), a = 0.2, b = 0.5$
- 6) $f(x) = \ln(1 + x) / x, a = 0.1, b = 1.0$
- 7) $f(x) = e^{-1.46x^2} / (3.5 + \sin(x)), a = 0.3, b = 0.8$
- 8) $f(x) = 1 / (\sqrt{x}(e^{0.9x} + 3)), a = 0.5, b = 2.0$
- 9) $f(x) = \sqrt{x(3 - x)} / (x + 1), a = 1.0, b = 1.2$
- 10) $f(x) = e^{1-x} / (2 + \sin(1 + x^2)), a = 0.4, b = 1.0$
- 11) $f(x) = \sin(x + 2) / (0.4 + \cos(x)), a = -1.0, b = 1.0$
- 12) $f(x) = (\sqrt{2 + x^2}) / ((1 + \cos(2x))\sqrt{1 - x^2}), a = 0.0, b = 1.0$
- 13) $f(x) = (x^2 + \sin(0.48(x + 2)) / e^{(x^2+0.38)}), a = 0.4, b = 1.0$
- 14) $f(x) = x^4 / (0.5x^2 + x + 6), a = 0.4, b = 1.0$
- 15) $f(x) = x / \sin^3(2x), a = 0.1, b = 0.5$

5.2 Последовательности

- 1) Факториал. $n_i = n_{i-1} \cdot i, n_0 = 1$
- 2) Последовательность Фибоначчи. $F_n = F_{n-1} + F_{n-2}, n \geq 2, F_0 = 0, F_1 = 1$
- 3) Арифметическая прогрессия $a_n = a_{n-1} + d$
- 4) Геометрическая прогрессия $b_n = b_{n-1} \cdot q$
- 5) Возвратная последовательность $x_n = p \cdot x_{n-1} + q \cdot x_{n-2}$

Лабораторная работа № 3. Тестирование методом черного ящика

1 Цель работы

Изучить подход к тестированию методом черного ящика.

2 Краткая теория

Тестирование программного кода – процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под дефектом здесь понимается участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (т.е. поведению, не соответствующему требованиям).

2.1 Критерии тестирования

Рассмотрим требования к идеальному критерию тестирования:

- 1) Критерий должен быть достаточным, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.
- 2) Критерий должен быть полным, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.
- 3) Критерий должен быть надежным, т.е. любые два множества тестов, удовлетворяющие ему, одновременно должны раскрывать или не раскрывать ошибки программы
- 4) Критерий должен быть легко проверяемым, например вычисляемым на тестах

В общем случае не существует полного и надежного критерия, зависящего от программ или спецификаций. Поэтому на практике применяются методы, удовлетворяющие данным критериям частично, и, комбинируя их, можно достичь приемлемого результата.

Существуют несколько классов критериев тестирования:

- 1) Функциональные критерии формулируются в описании требований к программному изделию (критерии так называемого "черного ящика")
- 2) Структурные критерии используют информацию о структуре программы (критерии так называемого "белого ящика")
- 3) Критерии стохастического тестирования формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.

- 4) Мутационные критерии ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

2.2 Функциональные критерии тестирования

Функциональный критерий – важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, контроль степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением.

При тестировании с помощью модели «черного ящика» подразумевается, что тестировщик ничего не знает о внутреннем устройстве метода/модуля/библиотеки, а имеет лишь набор спецификаций, которым должен соответствовать код. На основе этих спецификаций выбираются характерные наборы данных, позволяющие определить наличие ошибки.

Проблема такого подхода заключается, прежде всего, в его трудоемкости, так как документы, фиксирующие требования к программному изделию, как правило, достаточно объемны, тем не менее, соответствующая проверка должна быть всеобъемлющей.

2.3 Стратегии тестирования с помощью метода «черного ящика»

Невозможность «полного» тестирования программы делает главной целью любой стратегии тестирования уменьшение этой «неполноты». Поэтому необходимо соблюдать некоторый набор правил для получения наиболее удачного варианта тестов.

При тестировании программного обеспечения наиболее часто используются два подхода: метод черного и метод белого ящика, которые представляют собой не просто два метода, а классы методов или стратегии.

Методологии тестирования можно разделить на две группы: стратегии черного ящика и стратегии белого ящика.

Стратегии черного ящика в свою очередь делятся на:

- эквивалентное разбиение;
- анализ граничных значений;
- применение функциональных диаграмм;
- предположение об ошибке.

Стратегии белого ящика подразделяются на:

- покрытие операторов;
- покрытие решений;
- покрытие условий;

- покрытие решений/условий.

Рассмотрим стратегии черного ящика.

2.3.1 Эквивалентное разбиение

Тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных и желательно выбрать для тестирования самое подходящее подмножество (т. е. подмножество с наивысшей вероятностью обнаружения большинства ошибок).

Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

- уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;
- покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения. Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем, чтобы минимизировать общее число необходимых тестов. Во-вторых, необходимо пытаться разбить входную область программы на конечное число классов эквивалентности так, чтобы можно было предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса.

Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки (в том случае, когда некоторое подмножество класса эквивалентности не попадает в пределы любого другого класса эквивалентности, так как классы эквивалентности могут пересекаться).

Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как эквивалентное разбиение. Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а первое – для разработки минимального набора тестов, покрывающих эти условия.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

- 1) выделение классов эквивалентности;
- 2) построение тестов.

2.3.1.1 Выделение классов эквивалентности

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп.

Различают два типа классов эквивалентности: правильные классы эквивалентности, представляющие правильные входные данные программы, и неправильные классы эквивалентности, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения). Таким образом, придерживаются одного из принципов тестирования о необходимости сосредоточивать внимание на неправильных или неожиданных условиях.

Если задаться входными или внешними условиями, то выделение классов эквивалентности представляет собой в значительной степени эвристический процесс. При этом существует ряд правил:

1. Если входное условие описывает *область* значений (например, «целое данное может принимать значения от 1 до 99»), то определяются один правильный класс эквивалентности ($1 \leq \text{значение целого данного} \leq 99$) и два неправильных (значение целого данного < 1 и значение целого данного > 99).
2. Если входное условие описывает *число* значений (например, «в автомобиле могут ехать от одного до шести человек»), то определяются один правильный класс эквивалентности и два неправильных (ни одного и более шести человек).
3. Если входное условие описывает *множество* входных значений и есть основание полагать, что каждое значение программа трактует особо (например, «известны должности ИНЖЕНЕР, ТЕХНИК, НАЧАЛЬНИК ЦЕХА, ДИРЕКТОР»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс эквивалентности (например, «БУХГАЛТЕР»).
4. Если входное условие описывает *ситуацию* «должно быть» (например, «первым символом идентификатора должна быть буква»), то определяется один правильный класс эквивалентности (первый символ – буква) и один неправильный (первый символ – не буква).
5. Если есть любое основание считать, что различные элементы класса эквивалентности *трактуются* программой *неодинаково*, то данный класс эквивалентности разбивается на меньшие классы эквивалентности.

2.3.1.2 Построение тестов

Второй шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.
2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор, пока все правильные классы эквивалентности не будут покрыты (только не общими) тестами.
3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы эквивалентности не будут покрыты тестами.

Причина покрытия неправильных классов эквивалентности индивидуальными тестами состоит в том, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

2.3.2 Анализ граничных значений

Как показывает опыт, тесты, исследующие граничные условия, приносят большую пользу, чем тесты, которые их не исследуют. Граничные условия – это ситуации, возникающие непосредственно *на, выше* или *ниже* границ входных и выходных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.

2. При разработке тестов рассматривают не только входные условия (пространство входов), но и пространство результатов (т. е. выходные классы эквивалентности).

Достаточно трудно описать принимаемые решения при анализе граничных значений, так как это требует определенной степени творчества и специализации в рассматриваемой проблеме. (Следовательно, анализ граничных значений, как и многие другие аспекты тестирования, в значительной мере основывается на способностях человеческого интеллекта.)

Тем не менее, существует несколько общих правил этого метода.

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений. Например,

если правильная область входных значений есть от -1.0 до $+1.0$, то нужно написать тесты для ситуаций -1.0 , 1.0 , -1.001 и 1.001 .

2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то получить тесты для 0, 1, 255 и 256 записей.

3. Использовать первое правило для каждого выходного условия. Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет \$0.00, а максимум – \$1165.25, то построить тесты, которые вызывают расходы с \$0.00 и \$1165.25. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 1165.25 дол. Заметим, что важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей (например, при рассмотрении подпрограммы вычисления синуса). Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность.

4. Использовать второе правило для каждого выходного условия. Например, если система информационного поиска отображает на экране наиболее релевантные статьи в зависимости от входного запроса, но никак не более четырех рефератов, то построить тесты, такие, чтобы программа отображала нуль, один и четыре рефера, и тест, который мог бы вызвать выполнение программы с ошибочным отображением пяти рефератов.

5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

6. Попробовать свои силы в поиске других граничных условий.

Тем не менее, и здесь есть вероятность пропуска ошибки. Таким образом, существенное различие между анализом граничных значений и эквивалентным разбиением заключается в том, что анализ граничных значений исследует ситуации, возникающие *на* и *вблизи* границ эквивалентных разбиений.

2.4 Пример построения теста методом «черного ящика»

Задача: необходимо протестировать метод, который получает в качестве параметров два целых числа x и y , интерпретируемые как координаты точки на декартовой плоскости.

Выходом метода является номер четверти q , в которой находится точка. В остальных случаях возвращается «0».

Определим тестовые требования к методу:

1. Проверка возврата корректных значений для всех четвертей.
2. Проверка возврата корректных значений для координатных осей.
3. Проверка работы при передаче ошибочных значений

Примечание.

Ошибканые варианты в данном конкретном случае в тестовый набор не включаются, так как любые варианты входных параметров (целочисленных координат) является допустимым, а для статически типизированных языков использование несопоставимых типов данных в качестве параметров метода исключаются на этапе компиляции.

В остальных случаях ошибочные варианты должны обязательно присутствовать в тестах.

Рассмотрим пример построения тестов при помощи анализа граничных значений.

Первому тестовому требованию будет соответствовать выделение классов эквивалентности по выходным значениям для каждой четвертей (пространство результатов).

Второй пункт соответствует проверке граничных значений, который будут соответствовать неправильным классам эквивалентности. При этом граничные условия смежных четвертей могут быть проверены одним тестом (т.е. при проверке границ первой четверти, одни и те же точки одновременно проверяют границы I и IV четвертей). Таким образом, набор тестов можно сократить до следующего набора классов эквивалентности:

1. Точка в I четверти.
2. Точка в II четверти.
3. Точка в III четверти.
4. Точка в IV четверти.
5. Точка между I и II четвертью (точки на оси OY (верхняя полуось)).
6. Точка между I и IV четвертью (точки на оси OX (правая полуось)).
7. Точка между II и III четвертью (точки на оси OX (левая полуось)).
8. Точка между III и IV четвертью (точки на оси OY (нижняя полуось)).
9. Точка между II и IV четвертью (точка в центре координат).
10. Точка между I и III четвертью (точка в центре координат).

Примечание.

Если бы мы использовали только классы эквивалентности на основе входных данных, то они бы соответствовали различным вариантам

комбинаций входных значений (различных вариантов знаков x и y $[x>0;x=0;x<0;y>0;y=0;y<0]$). И тогда набор тестов был бы следующим:

1. $x>0;y>0$ - (точка в I четверти)
2. $x>0;y=0$ - (точка между I и IV четвертью)
3. $x>0;y<0$ - (точка в IV четверти)
4. $x=0;y>0$ - (точка между I и II четвертью)
5. $x=0;y=0$ - (точка между II и IV четвертью, I и III четвертью)
6. $x=0;y<0$ - (точка между III и IV четвертью)
7. $x<0;y>0$ - (точка в II четверти)
8. $x<0;y=0$ - (точка между II и III четвертью)
9. $x<0;y<0$ - (точка в III четверти)

Как видно мы получили почти тот же результат, хотя для других задач наборы классов эквивалентности могут существенно различаться.

В итоге получим следующий набор тестовых случаев (таблица 2.1). После номера теста в таблице идут параметры метода, а в правом столбце указаны покрываемые тестом классы эквивалентности.

Таблица 3.1 – Тестовый набор

№ теста	x	y	q	К.Э.
1	1	9	1	1
2	-3	2	2	2
3	-2	-1	3	3
4	3	-2	4	4
5	21	0	0	6
6	-3	0	0	7
7	0	-1	0	8
8	0	2	0	5
9	0	0	0	9, 10

На этих основе тестов проверяется наличие ошибок в работе метода.

3 Контрольные вопросы

- 1) В чем заключается суть тестирования с помощью модели «чёрного ящика»?
- 2) Какие стратегии метода «чёрного ящика» вы знаете?
- 3) Что такое класс эквивалентности?
- 4) Опишите правила построения классов эквивалентности?
- 5) Как осуществляется анализ граничных значений?

- 6) Опишите правила построения тестов на основе анализа граничных значений?
- 7) В чем достоинство использования функциональных диаграмм?

4 Задание

- 1) Создать класс (в соответствии с вариантом задания из п.5), реализующий проверку принадлежности точки различным областям плоскости, задаваемых пересечением фигур.
- 2) Выделить классы эквивалентности.
- 3) Проверить граничные значения для каждого из классов эквивалентности.
- 4) Составить наборы тестовых данных для созданного метода.
- 5) Протестировать метод на основе тестового набора с использованием программных отладочных средств.
- 6) Составить отчет о результатах проведенного тестирования.

5 Варианты заданий

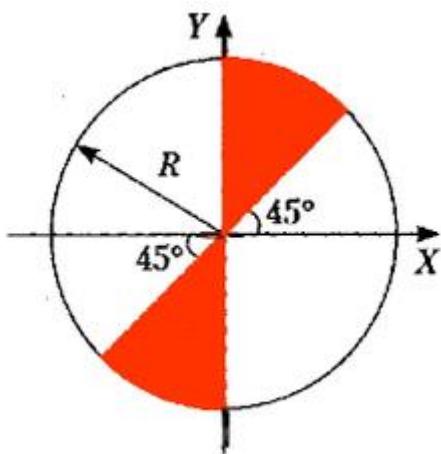
Создать класс, реализующий проверку принадлежности точки различным областям плоскости, задаваемых пересечением фигур.

Значения, характеризующие фигуры (R , a , b , и т.д.), задать как свойства классов.

В составе класса создать метод `TestPoint`, определяющий в какой из областей, задаваемых представленными ниже фигурами, находится точка с заданными координатами. На вход передаются координаты точки. Результатом работы метода должен быть номер области.

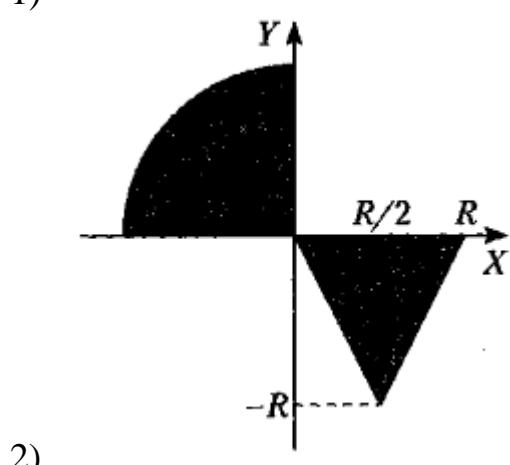
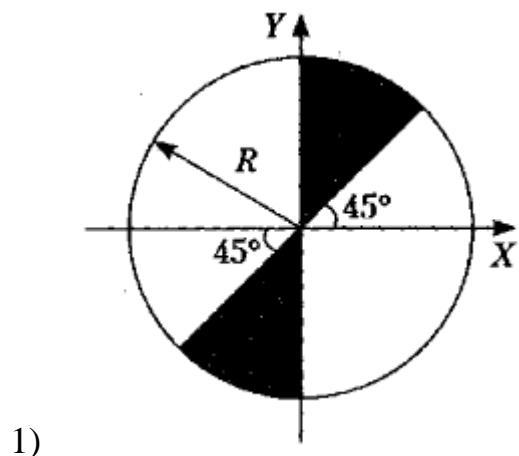
Нумерация областей производится по усмотрению разработчика, однако должно сохраняться их соответствие между запусками программы. При разбиении на области **не учитывать** разделение координатными осями.

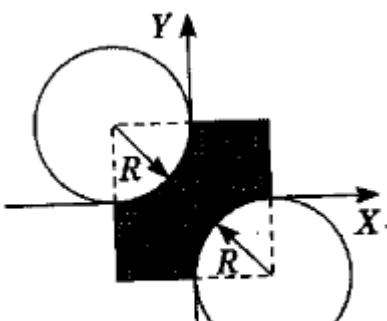
Например, для приведенного ниже варианта области будут следующими:



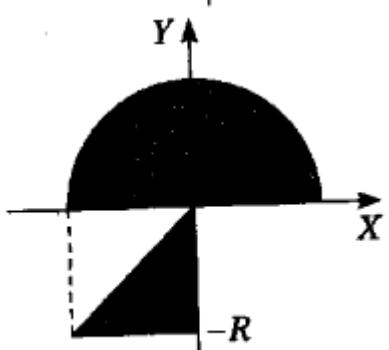
1. Закрашенная область внутри окружности (включая точки на линиях разделения закрашенной и не закрашенной областей внутри окружности) (красный цвет).
2. Не закрашенная область внутри окружности (включая точки окружности (черного цвета)).
3. Внешняя область окружности.

Варианты заданий:

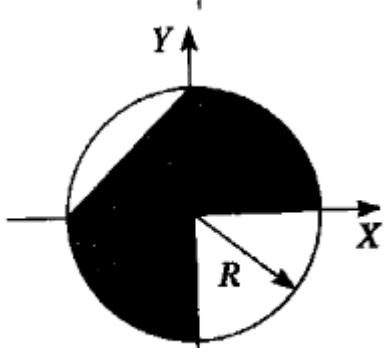




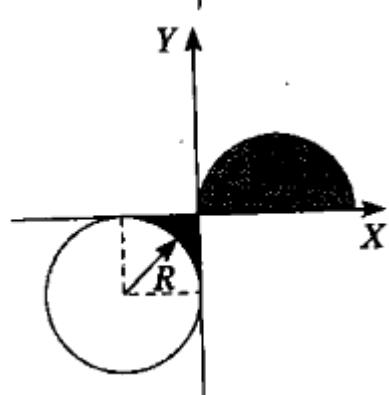
3)



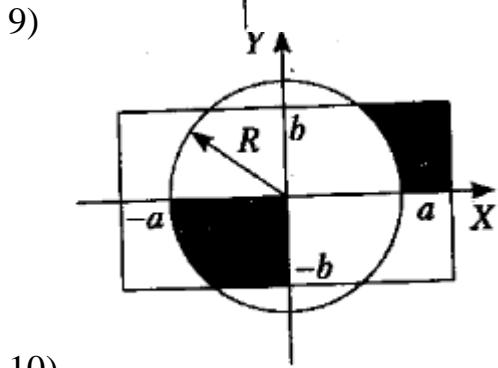
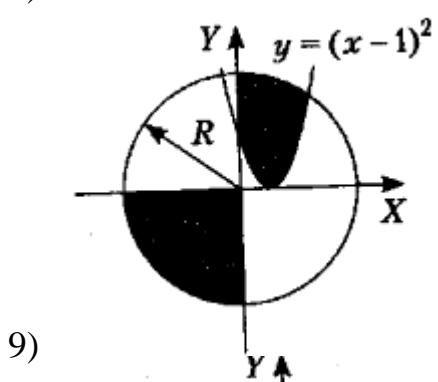
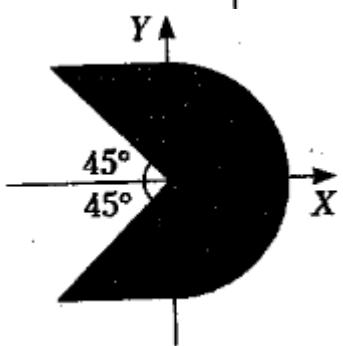
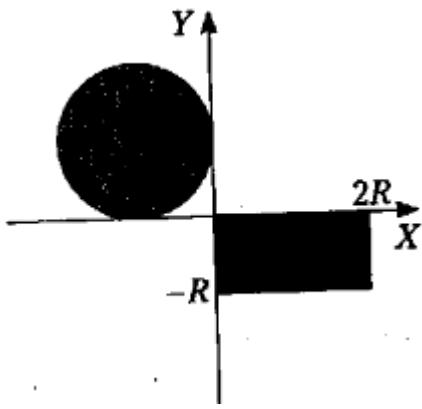
4)

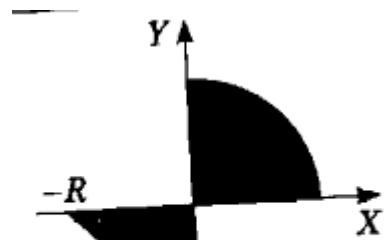


5)

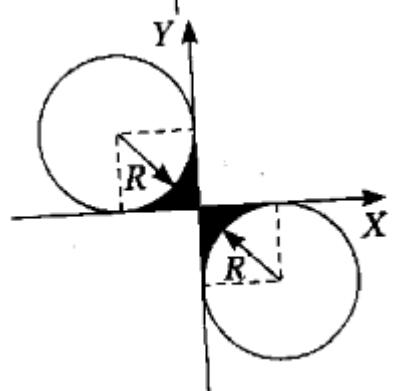


6)

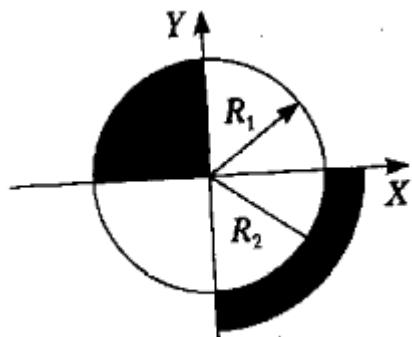




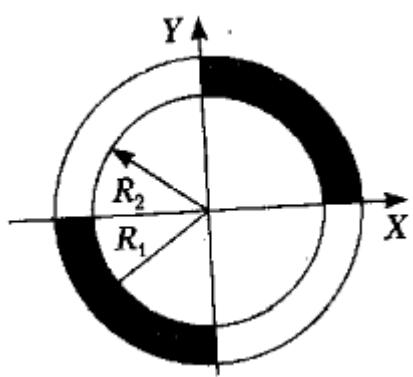
11)



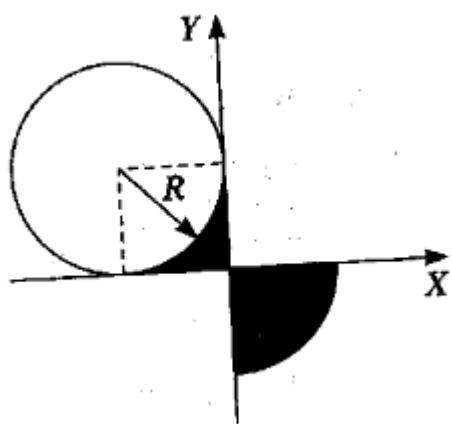
12)



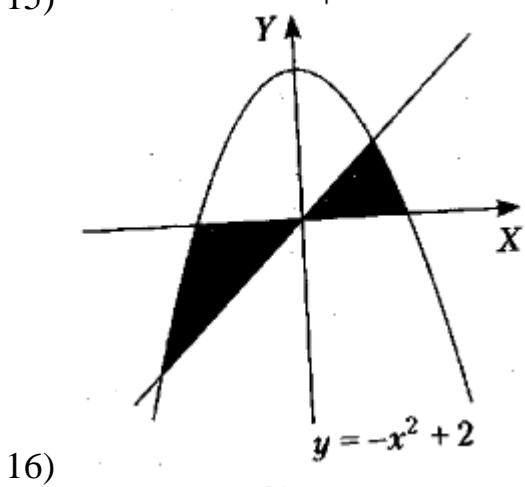
13)



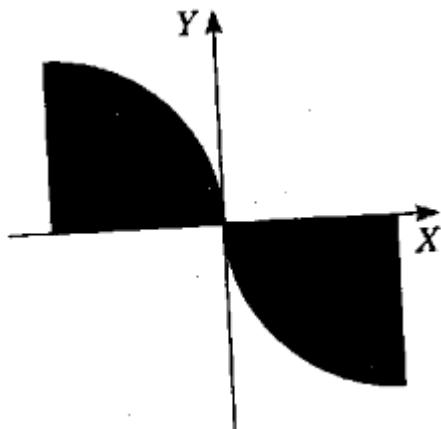
14)



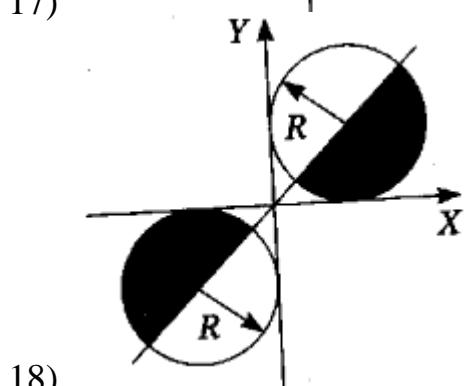
15)



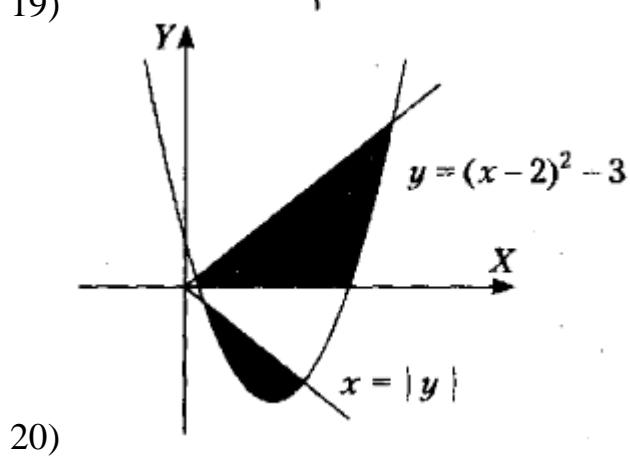
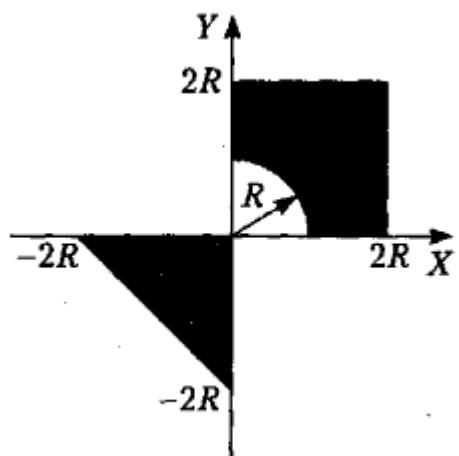
16)



17)



18)



Лабораторная работа № 4. Тестирование методом белого ящика

1 Цель работы

Изучить подход к тестированию методом белого ящика.

2 Краткая теория

2.1 Понятие тестирования. Цели и задачи тестирования

Тестирование программного кода – процесс выполнения программного кода, направленный на выявление существующих в нем дефектов. Под дефектом здесь понимается участок программного кода, выполнение которого при определенных условиях приводит к неожиданному поведению системы (т.е. поведению, не соответствующему требованиям).

Задача тестирования – определение условий, при которых проявляются дефекты системы и протоколирование этих условий. В задачи тестирования обычно не входит выявление конкретных дефектных участков программного кода и никогда не входит исправление дефектов – это задача отладки, которая выполняется по результатам тестирования системы.

Поскольку современные программные системы имеют весьма значительные размеры, при тестировании их программного кода используется метод функциональной декомпозиции. Система разбивается на отдельные модули (классы, пространства имен и т.п.), имеющие определенную требованиями функциональность и интерфейсы. После этого по отдельности тестируется каждый модуль – выполняется модульное тестирование.

Затем выполняется сборка отдельных модулей в более крупные конфигурации – выполняется интеграционное тестирование, и наконец, тестируется система в целом – выполняется системное тестирование.

Типичная процедура тестирования состоит в подготовке и выполнении тестовых примеров (также называемых просто тестами). Каждый тестовый пример проверяет одну «ситуацию» в поведении модуля и состоит из списка значений, передаваемых на вход модуля, описания запуска и выполнения переработки данных – тестового сценария, и списка значений, которые ожидаются на выходе модуля в случае его корректного поведения. Тестовые сценарии составляются таким образом, чтобы исключить обращения к внутренним данным модуля, все взаимодействие должно происходить только через его внешние интерфейсы.

Выполнение тестового примера поддерживается тестовым окружением, которое включает в себя программную реализацию тестового

сценария. Выполнение начинается с передачи модулю входных данных и запуска сценария. Реальные выходные данные, полученные от модуля в результате выполнения сценария сохраняются и сравниваются с ожидаемыми. В случае их совпадения тест считается пройденным, в противном случае – не пройденным. Каждый не пройденный тест указывает либо на дефект в тестируемом модуле, либо в тестовом окружении, либо в описании теста.

При тестировании часто бывает необходимо учитывать не только требования к системе, но и структуру программного кода тестируемого модуля. В этом случае тесты составляются таким образом, чтобы детектировать типичные ошибки программистов, вызванные неверной интерпретацией требований. Применяются проверки граничных условий, проверки классов эквивалентности. Отсутствие в системе возможностей, не заданных требованиями, гарантируют различные оценки покрытия программного кода тестами, т.е. оценки того, какой процент тех или иных языковых конструкций выполнен в результате выполнения всех тестовых примеров.

Рассмотрим стратегии «белого ящика» и структурные критерии подробней.

2.2 Структурные критерии

Структурные критерии используют модель программы в виде «белого ящика», что предполагает знание исходного текста программы или спецификации программы в виде потокового графа управления. Структурная информация понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный класс критериев часто используется на этапах модульного и интеграционного тестирования (Unit testing, Integration testing).

Структурные критерии базируются на основных элементах управляющих графов программы (УГП), операторах, ветвях и путях.

- Условие критерия **тестирования команд** (критерий С0) – набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.
- Условие критерия **тестирования ветвей** (критерий С1) – набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный

критерий часто используется в системах автоматизации тестирования.

– Условие критерия **тестирования путей** (критерий С2) – набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или числом классов выходных путей).

2.3 Стратегии тестирования метода «белого ящика»

Тестирование по принципу белого ящика характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Исчерпывающее тестирование по принципу белого ящика предполагает выполнение каждого пути в программе, но поскольку в программе с циклами выполнение каждого пути обычно нереализуемо, то тестирование всех путей не рассматривается.

При подходе «Белый ящик»/«Стеклянный ящик» тестировщик основывает идеи для тестирования на знании об устройстве и логике тестируемой части программы. Сценарии тестирования создаются с мыслью о том, чтобы протестировать определенную часть кода, а не определенный вариант поведения пользователя.

Структурные критерии не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию С2 мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

При тестировании системы, как стеклянного ящика, тестировщик имеет доступ не только к требованиям на систему, ее входам и выходам, но и к ее внутренней структуре – видит ее программный код.

Доступность программного кода расширяет возможности тестировщика тем, что он может видеть соответствие требований участкам программного кода и видеть тем самым – на весь ли программный код существуют требования. Программный код, для которого отсутствуют требования, называют кодом, непокрытым требованиями. Такой код является потенциальным источником неадекватного поведения системы. Кроме того, прозрачность системы позволяет углубить анализ ее участков, вызывающих проблемы – часто одна проблема нейтрализует другую и они никогда не возникают одновременно.

2.3.1 Покрытие операторов

Если отказаться полностью от тестирования всех путей, то можно показать, что критерием покрытия является выполнение каждого оператора программы, по крайней мере, один раз. Это метод покрытия операторов. К сожалению, это слабый критерий, так как выполнение каждого оператора, по крайней мере, один раз есть необходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика.

Предположим, что на рисунке 4.1 представлена небольшая программа, которая должна быть протестирована.

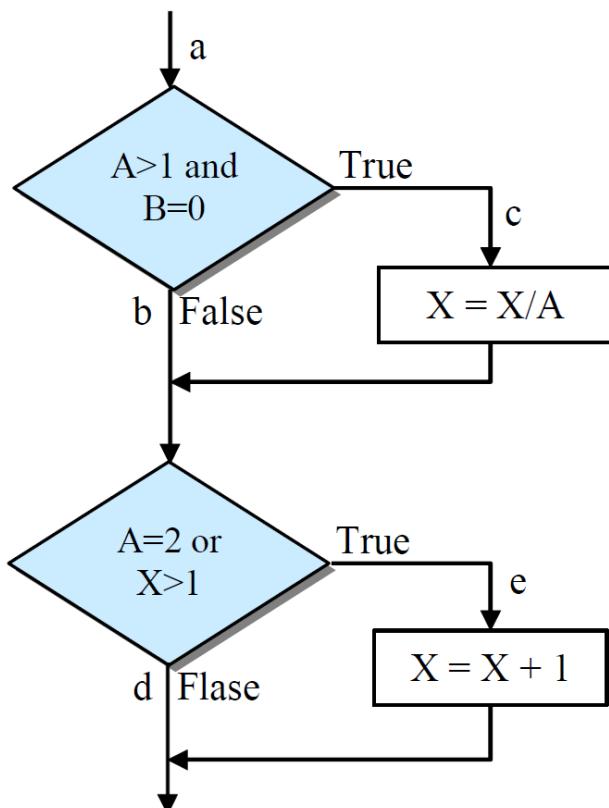


Рисунок 4.1 – Блок-схема тестируемой программы

Можно выполнить каждый оператор, записав один-единственный тест, который реализовал бы путь «асе».

Пусть второе решение записано в программе как $X > 0$ (во втором операторе условия); эта ошибка также не будет обнаружена. Кроме того, существует путь, в котором X не изменяется (путь «abd»). Если здесь ошибка, то и она не будет обнаружена. Таким образом, критерий покрытия операторов является настолько слабым, что его обычно не используют.

2.3.2. Покрытие решений

Более сильный критерий покрытия логики программы (и метод тестирования) известен как покрытие решений, или покрытие переходов. Согласно данному критерию должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение истина и ложь, по крайней мере, один раз. Иными словами, каждое направление перехода должно быть реализовано, по крайней мере, один раз. Примерами операторов перехода или решений являются операторы `while` или `if`.

Можно показать, что покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен. Однако существует, по крайней мере, три исключения. Первое – патологическая ситуация, когда программа не имеет решений. Второе встречается в программах или подпрограммах с несколькими точками входа (например, в программах на языке Ассемблера); данный оператор может быть выполнен только в том случае, если выполнение программы начинается с соответствующей точки входа. Третье исключение – операторы внутри `switch`-конструкций; выполнение каждого направления перехода не обязательно будет вызывать выполнение всех `case`-единиц. Так как покрытие операторов считается необходимым условием, покрытие решений, которое представляется более сильным критерием, должно включать покрытие операторов. Следовательно, покрытие решений требует, чтобы каждое решение имело результатом значения истина и ложь и при этом каждый оператор выполнялся бы, по крайней мере, один раз. Альтернативный и более легкий способ выражения этого требования состоит в том, чтобы каждое решение имело результатом значения истина и ложь и что каждой точке входа (включая каждую `case`-единицу) должно быть передано управление при вызове программы, по крайней мере, один раз.

Изложенное выше предполагает только двузначные решения или переходы и должно быть модифицировано для программ, содержащих многозначные решения (как для `case`-единиц). Критерием для них является выполнение каждого возможного результата всех решений, по крайней мере, один раз и передача управления при вызове программы или подпрограммы каждой точке входа, по крайней мере, один раз.

В программе, представленной на рисунке 3.1, покрытие решений может быть выполнено двумя тестами, покрывающими либо пути «`ace`» и «`abd`», либо пути «`acd`» и «`abe`». Если мы выбираем последнее

альтернативное покрытие, то входами двух тестов являются $A = 3, B = 0, X = 3$ и $A = 2, B = 1, X = 1$.

2.3.3 Покрытие условий

Покрытие решений – более сильный критерий, чем покрытие операторов, но и он имеет свои недостатки. Например, путь, где X не изменяется (если выбрано первое альтернативное покрытие), будет проверен с вероятностью 50 %. Если во втором решении существует ошибка (например, $X < 1$ вместо $X > 1$), то ошибка не будет обнаружена двумя тестами предыдущего примера.

Лучшим критерием (и методом) по сравнению с предыдущим является покрытие условий. В этом случае записывают число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз. Поскольку, как и при покрытии решений, это покрытие не всегда приводит к выполнению каждого оператора, к критерию требуется дополнение, которое заключается в том, что каждой точке входа в программу или подпрограмму, а также switch-единицам должно быть передано управление при вызове, по крайней мере, один раз.

Программа на рисунке 3.1 имеет четыре условия: $A > 1, B = 0, A = 2$ и $X > 1$.

Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где $A > 1, A \leq 1, B = 0$ и $B \neq 0$ в точке «*a*» и $A = 2, A \neq 2, X > 1$ и $X \leq 1$ в точке «*b*». Тесты, удовлетворяющие критерию покрытия условий, и соответствующие им пути:

1. $A = 2, B = 0, X = 4$ «*ace*».
2. $A = 1, B = 1, X = 1$ «*abd*».

Заметим, что, хотя аналогичное число тестов для этого примера уже было создано, покрытие условий обычно лучше покрытия решений, поскольку оно может (но не всегда) вызвать выполнение решений в условиях, не реализуемых при покрытии решений.

Хотя применение критерия покрытия условий на первый взгляд удовлетворяет критерию покрытия решений, это не всегда так. Если тестируется решение $\text{if}(A \&\& B)$, то при критерии покрытия условий требовались бы два теста – A есть истина, B есть ложь и A есть ложь, B есть истина. Но в этом случае не выполнялось бы тело условия. Тесты критерия покрытия условий для ранее рассмотренного примера покрывают результаты всех решений, но это только случайное совпадение. Например, два альтернативных теста:

1. $A = 1, B = 0, X = 3$.
2. $A = 2, B = 1, X = 1$,

покрывают результаты всех условий, но только два из четырех результатов решений (они оба покрывают путь «*аве*» и, следовательно, не выполняют результат истина первого решения и результат ложь второго решения).

2.3.4 Покрытие решений/условий

Очевидным следствием из этой дилеммы является критерий, названный покрытием решений/условий. Он требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия в решении, все результаты каждого решения выполнялись, по крайней мере, один раз и каждой точке входа передавалось управление, по крайней мере, один раз.

Недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий; часто подобное выполнение имеет место вследствие того, что определенные условия скрыты другими условиями. В качестве примера рассмотрим приведенную на рисунке 4.2 схему передач управления в коде, генерируемым компилятором языка, программы рисунку 4.1.

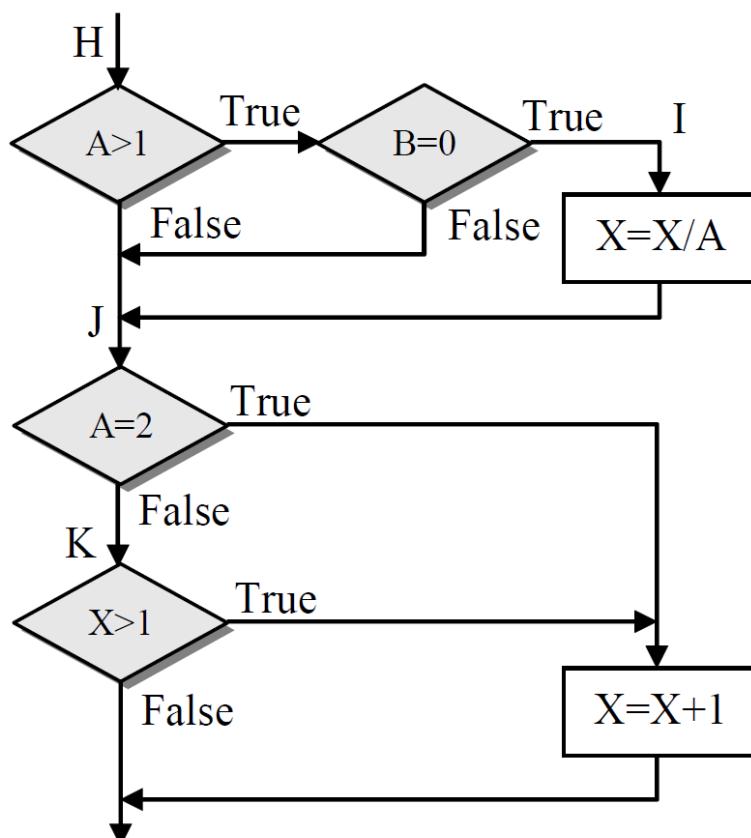


Рисунок 4.2 – Блок-схема тестируемой программы

Многоусловные решения исходной программы здесь разбиты на отдельные решения и переходы, поскольку большинство компьютеров не имеет команд, реализующих решения со многими исходами. Наиболее полное покрытие тестами в этом случае осуществляется таким образом, чтобы выполнялись все возможные результаты каждого простого решения.

Два предыдущих теста критерия покрытия решений не выполняют этого; они недостаточны для выполнения результата ложь решения Н и результата истина решения К. Набор тестов для критерия покрытия условий такой программы также является неполным; два теста (которые случайно удовлетворяют также и критерию покрытия решений/условий) не вызывают выполнения результата ложь решения I и результата истина решения К.

Причина этого заключается в том, что, как показано на рисунке 3.2, результаты условий в выражениях и и или могут скрывать и блокировать действие других условий. Например, если условие и есть ложь, то никакое из последующих условий в выражении не будет выполнено. Аналогично если условие или есть истина, то никакое из последующих условий не будет выполнено. Следовательно, критерии покрытия условий и покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

2.3.5 Комбинаторное покрытие условий

Критерием, который решает эти и некоторые другие проблемы, является комбинаторное покрытие условий. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись, по крайней мере, один раз.

По этому критерию для программы на рисунке 3.1 должны быть покрыты тестами следующие восемь комбинаций:

1. $A > 1, B = 0$.
2. $A > 1, B \neq 0$.
3. $A \leq 1, B = 0$.
4. $A \leq 1, B \neq 0$.
5. $A = 2, X > 1$.
6. $A = 2, X \leq 1$.
7. $A \neq 2, X > 1$.
8. $A \neq 2, X \leq 1$.

Заметим, что комбинации 5–8 представляют собой значения второго оператора if. Поскольку X может быть изменено до выполнения этого оператора, значения, необходимые для его проверки, следует

восстановить, исходя из логики программы с тем, чтобы найти соответствующие входные значения.

Для того чтобы протестировать эти комбинации, необязательно использовать все восемь тестов. Фактически они могут быть покрыты четырьмя тестами. Приведем входные значения тестов и комбинации, которые они покрывают:

A = 2, B = 0, X = 4 покрывает 1, 5;
A = 2, B = 1, X = 1 покрывает 2, 6;
A = 1, B = 0, X = 2 покрывает 3, 7;
A = 1, B = 1, X = 1 покрывает 4, 8.

То, что четырем тестам соответствуют четыре различных пути на рисунке 3.1, является случайным совпадением. На самом деле представленные выше тесты не покрывают всех путей, они пропускают путь «acd». Например, требуется восемь тестов для тестирования следующей программы:

```
if((x == y) && (z == 0) && end)
    j = 1;
else
    i = 1;
```

хотя она покрывается лишь двумя путями. В случае циклов число тестов для удовлетворения критерию комбинаторного покрытия условий обычно больше, чем число путей.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого:

1) вызывает выполнение всех результатов каждого решения, по крайней мере, один раз;

2) передает управление каждой точке входа (например, точке входа, case-единице) по крайней мере, один раз (чтобы обеспечить выполнение каждого оператора программы, по крайней мере, один раз).

Для программ, содержащих решения, каждое из которых имеет более одного условия, минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных комбинаций результатов условий в каждом решении и передающих управление каждой точке входа программы, по крайней мере, один раз. Слово «возможных» употреблено здесь потому, что некоторые комбинации условий могут быть нереализуемыми; например, в выражении $(a>2) \&\& (a<10)$ могут быть реализованы только три комбинации условий.

2.4 Пример построения теста методом «белого ящика»

Пусть имеется метод, который считает сумму положительных элементов целочисленного массива. Пример реализации такого метода приведён ниже.

Пример 3.1 – Метод нахождения суммы положительных элементов целочисленного массива

```
int SumPositive(int[] a)
{
    int sum = 0;           //1
    for (int i = 0;        //2
         i < a.Length;     //3
         i++)              //4
    {
        if (a[i] > 0)    //5
        {
            sum += a[i]; //6
        }
    }
    return sum;           //7
}
```

Данному методу будет соответствовать управляющий граф программы, представленный на рисунке 4.3.

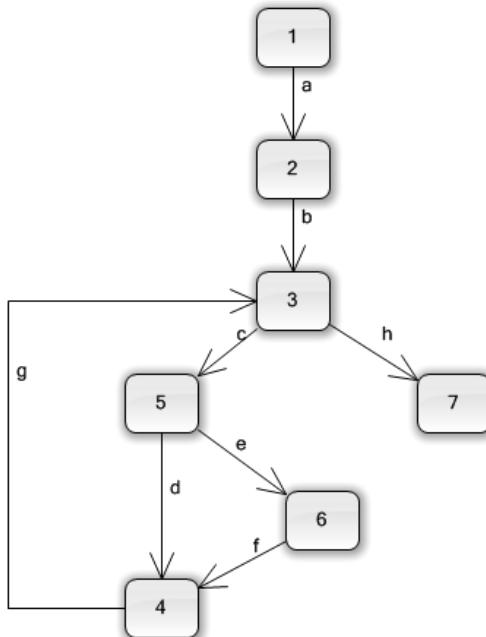


Рисунок 4.3 – Управляющий граф программы

Анализируя УГП, выбираем различные пути (с учетом классов путей) и для каждого из них подбираем входные данные (таблица 4.1).

Таблица 4.1 – Пути в УГП

№	Путь	a	Результат
1	(1,2,3,7) / abh	[]	0
2	(1,2,3,5,6,4,3,7) / abcdefgh	[1]	1
3	(1,2,3,5,4,3,7) / abcdgh	[-1]	0

Далее определяем, какие из критериев соответствуют полученным путям (таблица 4.2)

Таблица 4.2 – Соответствие критериев покрытия выбранным путям

№	Критерий	Пути
1	Покрытие операторов	2
2	Покрытие решений	1, 2, 3
3	Покрытие условий	1, 2, 3
4	Покрытие путей	1, 2, 3 (классов путей)

Проанализировав результаты, можно сделать вывод о достаточности набор тестовых данных (если каждый из критериев порывается хотя бы один из путей), или необходимо дополнить тестовый набор другими данными для реализации новых путей.

Рассмотрим еще один пример. Пусть имеется метод, проверяющий принадлежность точки некоторой области, представленной на рисунке 3.4.

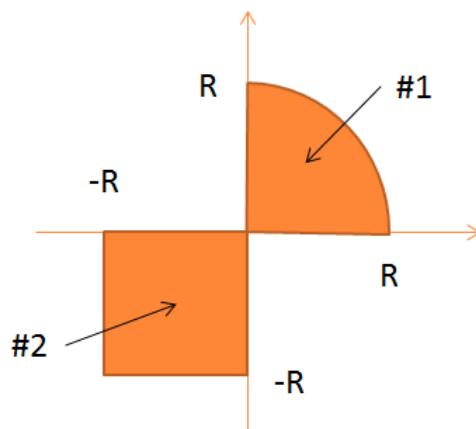


Рисунок 4.4 – Область тестового метода

Область состоит из двух частей, и метод должен возвращать номер области, в которой находится точка («1» или «2») и «-1» если точка не принадлежит области. Пример реализации такого метода приведён ниже.

Пример 4.2 – Метод проверки принадлежности области.

```
int Test(double x, double y)
{
    int result = -1;           //1
    if ((x >= 0) && (y >= 0)
        && (x*x + y*y <=R*R)) //2
    {
        result = 1;           //3
    }
    if ((x <= 0) && (x >= -R) &&
        (y <= 0) && (y >= -R)) //4
    {
        result = 2;           //5
    }
    return result;            //6
}
```

Данному методу будет соответствовать управляющий граф программы, представленный на рисунке 4.5.

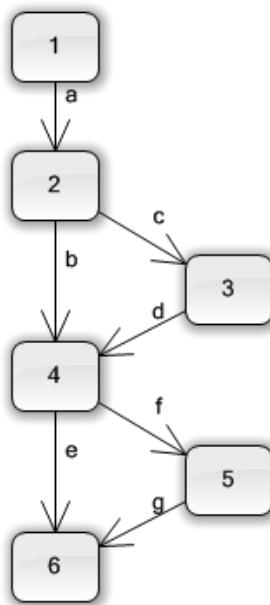


Рисунок 4.5 – Управляющий граф программы

Пусть $R=2$. Тогда исходя из УГП, подбираем входные данные таким образом, что бы были реализованы различные пути (с учетом классов путей) (таблица 4.3).

Таблица 4.3 – Пути в УГП

№	Путь	x	y	Результат
1	(1,2,3,4,6) / acde	1	1	1
2	(1,2,4,5,6) / abfg	-1	-1	2
3	(1,2,4,6) / abe	3	3	-1
4	(1,2,3,4,5,6) / acdfg	0	0	2

Далее определяем, какие из критериев соответствуют полученным путям (таблица 4.4)

Таблица 4.4 – Соответствие критериев покрытия выбранным путям

№	Критерий	Пути
1	Покрытие операторов	1, 2
2	Покрытие решений	1, 2
3	Покрытие условий	нет покрытия
4	Покрытие путей	1, 2, 3, 4

Из результатов таблицы 4.4, видно, что полученный тестовый набор не обеспечивает покрытие условий, т.е. в условном операторе потенциально может содержаться ошибка и желательно расширить тестовый набор дополнительными тестовыми данными.

3 Контрольные вопросы

- 1) Какова цель тестирования?
- 2) Какие классы тестовых критериев вы знаете?
- 3) Перечислите и охарактеризуйте структурные критерии?
- 4) В чем заключается суть тестирования с помощью модели «белого ящика»?
- 5) Какие стратегии тестирования методом белого ящика вы знаете?
- 6) В чем заключается суть метода покрытия операторов?
- 7) В чем заключается суть метода покрытия решений?
- 8) В чем заключается суть метода покрытия условий?
- 9) В чем заключается суть метода покрытия решений/условий?
- 10) В чем заключается суть комбинаторного покрытия условий?

4 Задание

- 1) Создать класс (в соответствии с вариантом задания из п.5), реализующий обработку с текстовой информацией.
- 2) Построить управляющий граф программы (УГП).
- 3) Используя знания о структуре программы определить набор тестов для тестирования методом белого ящика.
- 4) Указать какие пути реализуются (проверяются) тестом (см. таблица 4.1).

- 5) Указать какие критерии покрываются путем (см. таблица 4.2).
- 6) Оценить качество полученных тестовых данных с точки зрения покрытия кода тестами.
- 7) Протестировать метод с использованием средств автоматизации тестирования.
- 8) Составить отчет о результатах проведенного тестирования.

5 Варианты заданий

Создать класс, реализующий обработку с текстовой информацией.
Стандартные методы классов по обработке строк не использовать.

Варианты заданий:

1. Создать класс, реализующий разбиение текста на слова. Набор разделителей передавать в качестве параметра метода.
2. Создать класс, реализующий разбиение текста на предложения. Набор разделителей передавать в качестве параметра метода.
3. Создать класс, реализующий метод для слияния строк (посимвольное).
4. Создать класс, реализующий метод для вставки символа в произвольную позицию строки.
5. Создать класс, реализующий метод для вставки подстроки в произвольную позицию строки.
6. Создать класс, реализующий метод для удаления символов из заданного набора с начала строки (LTrim).
7. Создать класс, реализующий метод для удаления символов из заданного набора с конца строки (RTrim).
8. Создать класс, реализующий метод для удаления символов из заданного набора с обоих концов строки (Trim).
9. Создать класс, реализующий метод для замены первого вхождения символа в строке.
10. Создать класс, реализующий метод для подсчёта количества вхождений символа в строке.
11. Создать класс, реализующий метод для замены последнего вхождения символа в строке.
12. Создать класс, реализующий метод для замены всех вхождений символа в строке.
13. Создать класс, реализующий метод для замены подстроки в строке на другую подстроку.

Лабораторная работа № 5. Тестирование, управляемое данными (Data-Driven Unit Tests) и Анализ покрытия кода (Code Coverage)

1 Цель работы

Изучить подход к автоматизации процесса тестирования с помощью средств среды разработки Microsoft Visual Studio, а также понятие покрытия кода (Code Coverage) тестами и научиться использовать на практике средства автоматизации определения покрытия.

2 Краткая теория

2.1 Тестирование, управляемое данными

При тестировании часто возникает ситуация, при которой сам тестовый код практически не отличается для различных тестовых методов, за исключением тестовых данных, передаваемых на вход тестируемого метода и результирующего значения.

В таких случаях используется так называемое «Параметризованное тестирование», при котором наборы тестовых данных и результаты построчно группируются в таблицу, и каждая строка интерпретируется как отдельный тест.

В среде Visual Studio также имеется возможность поддержки такого рода тестов, называемая «Тестирование, управляемое данными (Data-Driven Unit Tests)», обладающая достаточно обширными возможностями.

Тестирование, управляемое данными это тестирование, выполняемое для каждого варианта тестовых наборов.

При этом источником данных могут быть значения, указанные в атрибуте TestCase или строки в таблице, указанной в качестве источника для теста в атрибуте TestCaseSource.

Рассмотрим оба варианта построения тестов.

Для начала создадим демонстрационный проект содержащий класс-утилиту с методом, выполняющим сложения двух чисел (пример 5.1).

Пример 5.1. Тестируемый класс

```
using System;

namespace STLabsExamples05x01_Project
{
    public static class Calc
    {
        public static int Sum(int a, int b)
        {
            return a + b;
        }
    }
}
```

```
    }  
}
```

Для задания тестового набора в первом случае необходимо в атрибутах TestCase указать тестовые наборы значений, а сам тестовый метод сделать параметризованным. При этом каждому параметру будет присваиваться значение соответствующего аргумента атрибута TestCase.

Пример 5.2. Модульное тестирование с помощью TestCase

```
using System;  
  
using NUnit.Framework;  
using STLabsExamples05x01_Project;  
  
namespace STLabsExamples05x01_ProjectTest  
{  
    [TestFixture]  
    public class CalcTestDDTTestCase  
    {  
        [TestCase(0, 0, 0)]  
        [TestCase(1, 0, 1)]  
        [TestCase(0, 1, 1)]  
        [TestCase(1, 1, 2)]  
        public void CalcSumTest(int a, int b, int result)  
        {  
            int actual = Calc.Sum(a, b);  
            Assert.AreEqual(result, actual);  
        }  
    }  
}
```

Второй вариант подразумевает использование некоторого внешнего по отношению к тесту источника для получения наборов тестовых данных. Для этого в атрибуте TestCaseSource указывается тип и источник данных для теста.

Допускается 3 формы использования атрибута TestCaseSource:

```
TestCaseSourceAttribute(Type sourceType); //устаревший вариант  
TestCaseSourceAttribute(Type sourceType, string sourceName);  
TestCaseSourceAttribute(string sourceName)
```

В качестве источника могут служить любые сущности обладающие следующими характеристиками:

- 1) Это должен быть метод, поле или свойство
- 2) Допускается использование, как статических компонентов, так и компонентов экземпляра.

- 3) Они должны возвращать IEnumerable или тип реализующий IEnumerable.
- 4) Возвращаемые значения должны быть совместимы с сигнатурой метода.

Варианты использования таких сущностей показаны в примере 5.3.

Пример 5.3. Модульное тестирование с помощью TestCaseSource

```

using System;
using System.Collections;

using NUnit.Framework;
using STLabsExamples05x01_Project;

namespace STLabsExamples05x01_ProjectTest
{
    [TestFixture]
    public class CalcTestDDTTestCaseSource
    {
        // Пример 1: Тестирование суммы
        // Работа через массив массивов тестовых данных
        [Test, TestCaseSource("SumCases")]
        public void CalcSumTestStatic(int a, int b, int result)
        {
            int actual = Calc.Sum(a, b);
            Assert.AreEqual(result, actual);
        }

        static object[] SumCases =
        {
            new object[] { 1, 1, 2 },
            new object[] { 3, 2, 5 },
            new object[] { 1, 4, 5 }
        };
    }

    // Пример 2: Четные числа
    // Работа через массив тестовых данных
    static int[] EvenNumbers = new int[] { 2, 4, 6, 8 };

    [Test]
    [TestCaseSource("EvenNumbers")]
    public void TestMethod(int num)
    {
        Assert.IsTrue(num % 2 == 0);
    }

    // Пример 3: Умножение чисел
    // Работа через метод генерирующий тестовые данные
    // Используется класс TestCaseData

```

```

// фреймворка NUnit
[Test, TestCaseSource("TestCaseMethod")]
public int MulTest(int n, int d)
{
    return n * d;
}

public static IEnumerable TestCaseMethod()
{
    yield return new TestCaseData(2, 3).Returns(6);
    yield return new TestCaseData(2, 2).Returns(4);
    yield return new TestCaseData(1, 8).Returns(8);
    yield return new TestCaseData(0, 1).Returns(0);
}

// Пример 4: Деление чисел
// Работа через типизированный класс
// генерирующий тестовые данные
// Используется класс TestCaseData
// фреймворка NUnit
[TestCaseSource(typeof(MyFactoryClass), "TestCases")]
public int DivideTest(int n, int d)
{
    return n / d;
}

public class MyFactoryClass
{
    public static IEnumerable TestCases
    {
        get
        {
            yield return new TestCaseData(12, 3).Returns(4);
            yield return new TestCaseData(12, 2).Returns(6);
            yield return new TestCaseData(12, 4).Returns(3);
            yield return new TestCaseData(0, 0)
                .Throws(typeof(DivideByZeroException))
                .SetName("DivideByZero")
                .SetDescription("An exception is expected");
        }
    }
}
}

```

2.2 Покрытие кода

2.2.1 Понятие покрытия кода

Покрытие кода (Code Coverage) - это мера, используемая при тестировании программного обеспечения, показывающая в процентах, насколько исходный код программы был протестирован.

Техника покрытия кода была одной из первых методик, изобретённых для систематического тестирования программного обеспечения.

2.2.2 Критерии покрытия

Существует несколько различных способов измерения покрытия, основные из них:

- покрытие операторов – каждая ли строка исходного кода была выполнена и протестирована;
- покрытие условий – каждая ли точка решения (вычисления истинно ли или ложно выражение) была выполнена и протестирована;
- покрытие путей – все ли возможные пути через заданную часть кода были выполнены и протестированы;
- покрытие функций – каждая ли функция программы была выполнена;
- покрытие вход/выход – все ли вызовы функций и возвраты из них были выполнены;
- покрытие значений параметров – все ли типовые и граничные значения параметров были проверены.

Для программ с особыми требованиями к безопасности часто требуется продемонстрировать, что тестами достигается 100 % покрытие для одного из критериев. Некоторые из приведённых критериев покрытия связаны между собой; например, покрытие путей включает в себя и покрытие условий и покрытие операторов.

Обычно исходный код снабжается тестами, которые регулярно выполняются. Полученный отчёт анализируется с целью выявить ни разу не выпаленные (а значит и не протестированные) области кода. После этого набор тестов обновляется, и пишутся тесты для непокрытых областей. Цель состоит в том, чтобы получить набор тестов для регрессионного тестирования, тщательно проверяющих весь код.

Степень покрытия кода обычно выражают в виде процента. Обычно употребляются фразы вида «протестировано 75% кода», смысл которых зависит от того какой критерий был использован. Например, 70% покрытия путей – это лучший результат, чем 70% покрытия операторов.

Связь значения покрытия кода и качества тестового набора не является однозначной. Оно отражает покрытие тестами методом белого ящика.

2.3 Средства автоматизации анализа покрытия кода

2.3.1 Анализ покрытия кода в среде JetBrains Rider

Среда разработки JetBrains Rider имеет в своем составе средства для анализа покрытия кода.

Для активации анализа покрытия кода необходимо в контекстном меню запуска тестов выбрать пункт «Cover Unit Tests» (рисунок 5.1)

После этого в окне «Code Coverage Results» можно получить результаты анализа покрытия кода. При этом в окне с кодом тестируемого компонента можно увидеть результаты покрытия каждой из строк (рисунок 5.2)

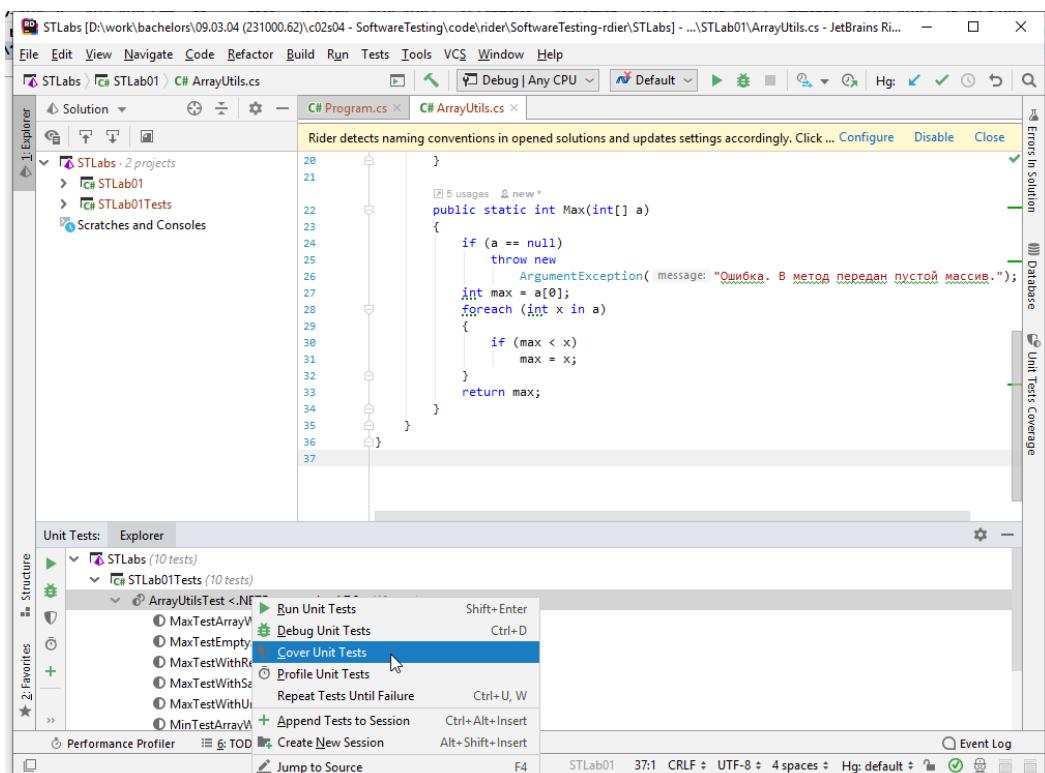


Рисунок 5.1 – Активация анализа покрытия кода

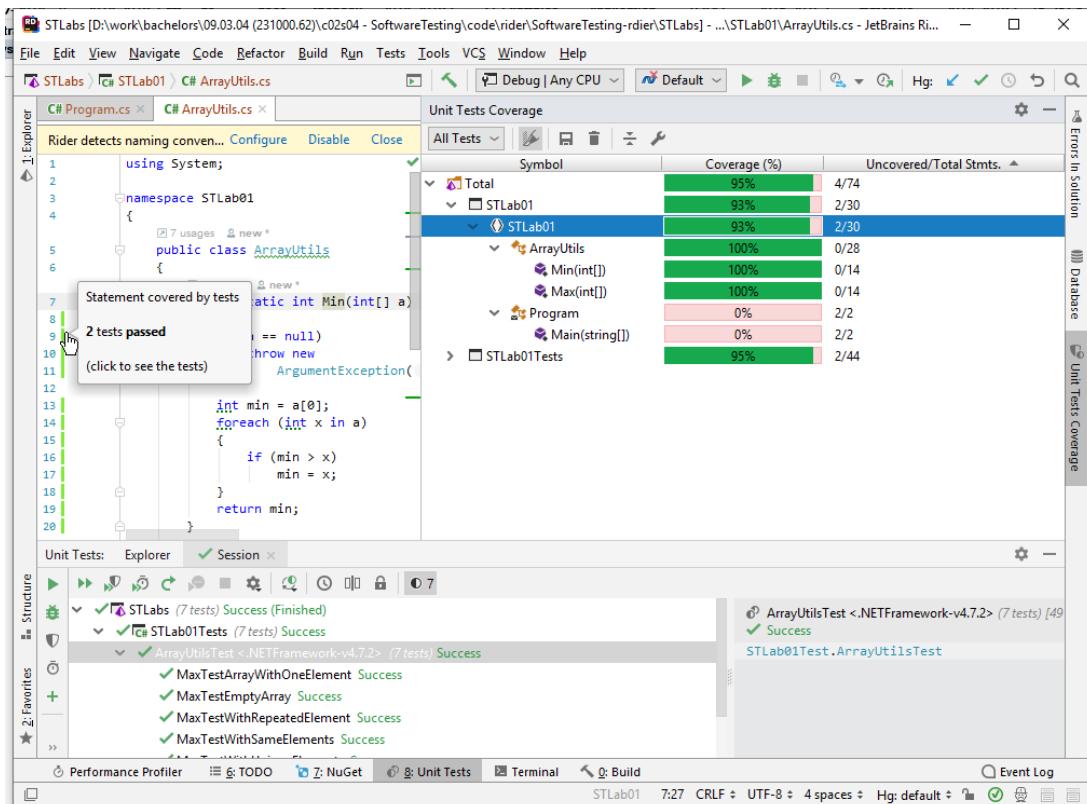


Рисунок 5.2 –Результаты анализа покрытия кода

2.3.2 Анализ покрытия кода в среде Visual Studio

Среда разработки Visual Studio (только в версии Enterprise/Ultimate) имеет в своем составе средства для анализа покрытия кода.

Для активации анализа покрытия кода необходимо выполнить ряд действий по настройке тестового проекта:

1. Открыть настройки local.testsettings (можно получить к ним доступ через меню Test -> Edit Test Settings -> Local (local.testsettings)).
2. Выбрать раздел «Data and Diagnostics».
3. Сделать активным пункт «Code Coverage» (рисунок 5.3)
4. После этого дважды кликнув на строку «Code Coverage» выбрать сборки (assemblies) для анализа (обычно исполняемый файл с реализацией тестируемого кода) (рисунок 5.4).

После этого запустив тесты можно получить результаты анализа покрытия кода. Для этого необходимо открыть окно результатов покрытия «Code Coverage Results» (рисунок 5.5).

Результаты покрытия кода и сами строки тестируемого кода отображаются в окне «Code Coverage Results» (рисунок 5.6).

Оценка покрытия осуществляется на уровне сборки (assembly), пространства имен (namespace), класса (class) и метода (method). По

умолчанию данные в окне «Code Coverage Results» отображают общее число и процент покрытых и непокрытых строк блоков.

При двойном клике на любой из тестируемых методов среда переходит к коду и отображает прорывание строк (рисунок 5.7). При этом покрытые строки будут выделены светло-голубым цветом, а непокрытые – красно-коричневым.

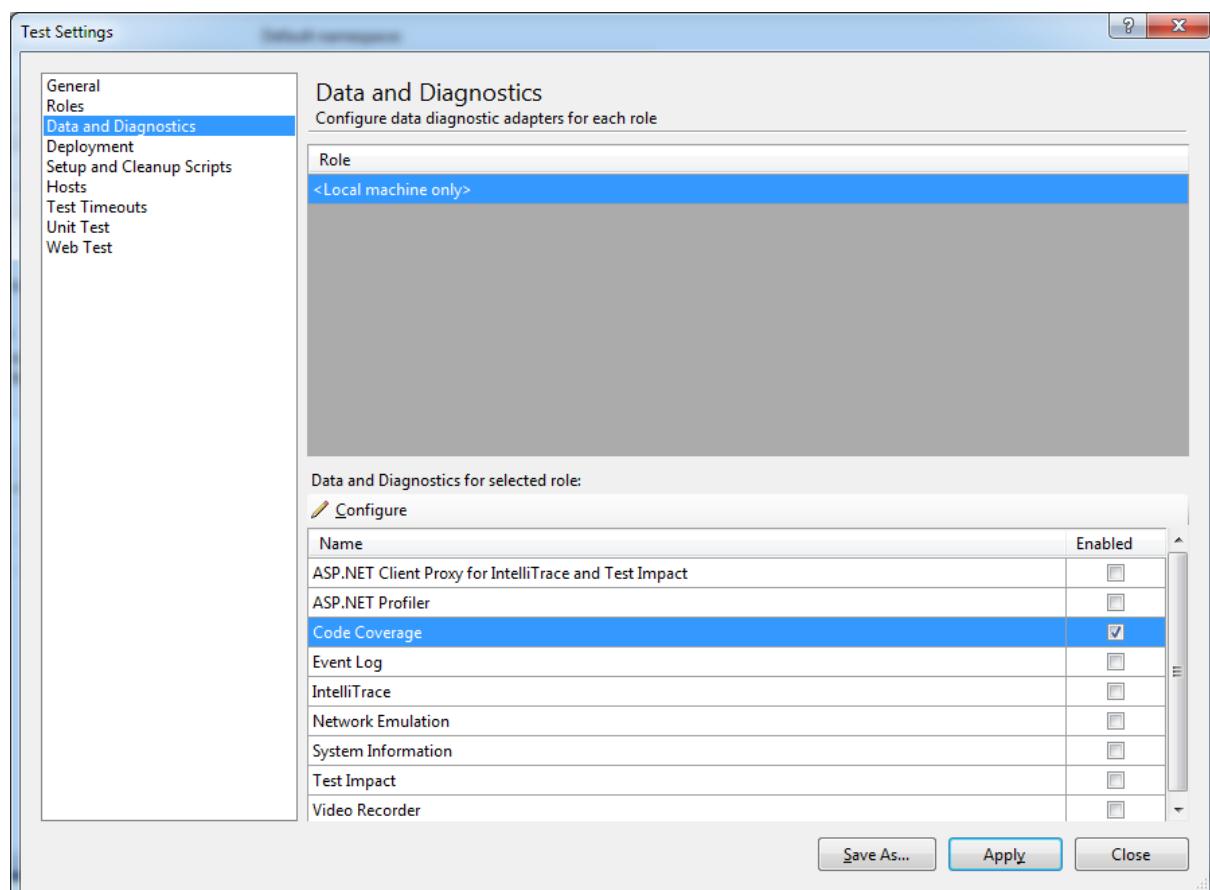


Рисунок 5.3 – Активация анализа покрытия кода

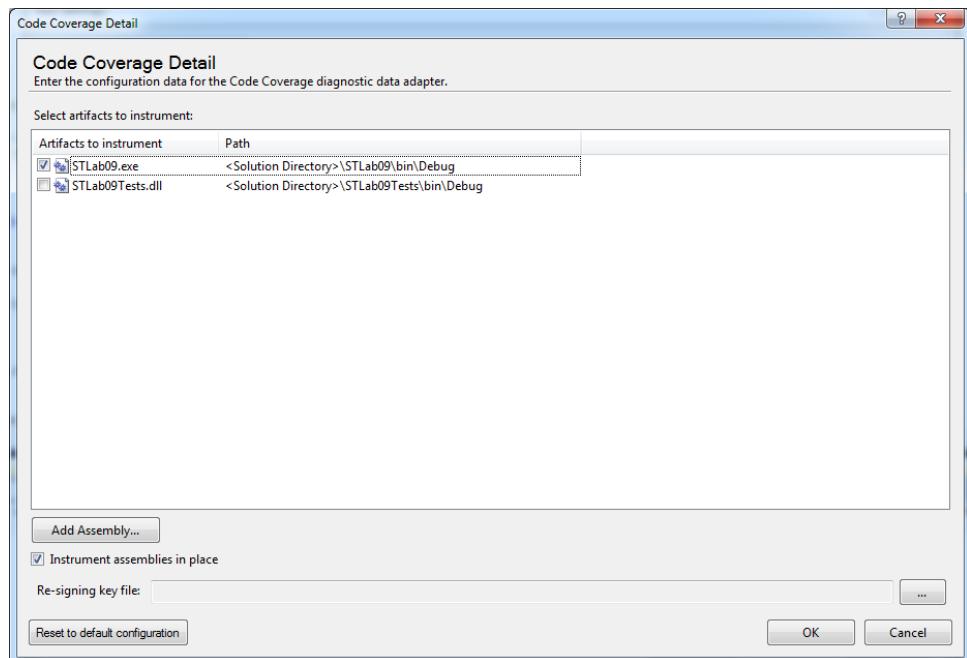


Рисунок 5.4 – Выбор сборки для анализа покрытия кода

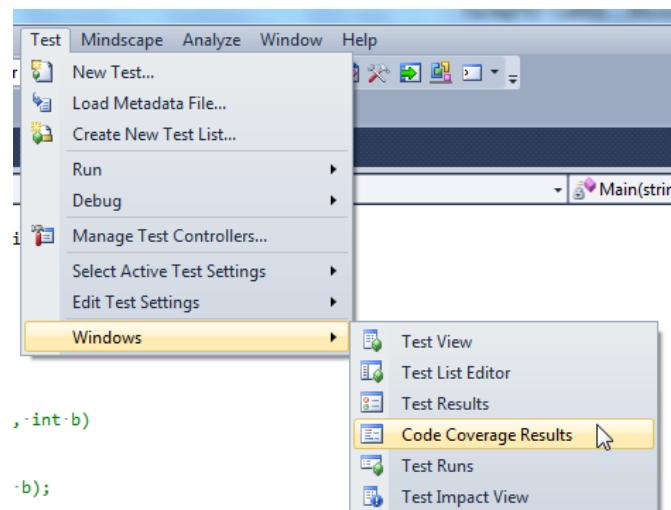


Рисунок 5.5 – Открытие окна результатов анализа покрытия кода

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
root@i7-3770 2013-04-25 01:02:46	2	10,53%	17	89,47%
STLab09.exe	2	10,53%	17	89,47%
{} STLab09	2	10,53%	17	89,47%
DayOfWeekConverter	1	5,56%	17	94,44%
cctor()	0	0,00%	9	100,00%
GetDayOfWeekName(int32)	1	11,11%	8	88,89%
Program	1	100,00%	0	0,00%
Main(string[])	1	100,00%	0	0,00%

Рисунок 5.6 – Результаты анализа покрытия кода

```

8     class DayOfWeekConverter
9     {
10         private static Dictionary<int, string> dict;
11         static DayOfWeekConverter()
12         {
13             dict = new Dictionary<int, string>();
14
15             dict[0] = "Sunday";
16             dict[1] = "Monday";
17             dict[2] = "Tuesday";
18             dict[3] = "Wednesday";
19             dict[4] = "Thursday";
20             dict[5] = "Friday";
21             dict[6] = "Saturday";
22         }
23
24         static public string GetDayOfWeekName(int dayNumber)
25         {
26             if (!dict.ContainsKey(dayNumber))
27                 throw new ArgumentException();
28
29             if ((dayNumber > 6) && (dayNumber > 7))
30                 return "";
31
32             return dict[dayNumber];
33         }
34
35     }

```

Рисунок 5.7 – Результаты анализа покрытия в исходном коде кода

Варианты результатов анализа представлены в таблице 5.1.

Таблица 5.1 – Доступные варианты результатов анализа

Значение	Описание
Covered (Lines)	Показывает общее число выполненных при тестировании строк кода
Not Covered (Lines)	Показывает общее число невыполненных при тестировании строк кода
Covered (% Lines)	Показывает процент выполненных при тестировании строк кода
Not Covered (% Lines)	Показывает процент невыполненных при тестировании строк кода
Covered (Blocks)	Показывает общее число выполненных при тестировании блоков кода
Not Covered (Blocks)	Показывает общее число невыполненных при тестировании блоков кода
Covered (% Blocks)	Показывает процент выполненных при тестировании блоков кода
Not Covered (% Blocks)	Показывает процент невыполненных при тестировании блоков кода
Partially Covered (Lines)	Показывает общее число частично выполненных строк при тестировании
Partially Covered (% Lines)	Показывает процент частично выполненных строк при тестировании

При запуске тестов, покрытие кода вычисляется для блоков, строк и частично покрытых строк (Partially Covered).

Блок кода представляет собой путь выполнения кода с единственным входом и выходом, т.е. набор операторов, выполняемых последовательно. Блок кода завершается либо ветвлением, либо вызовом функции, либо набором операторов связанных с обработкой исключений.

При покрытии строк кода учитываются только исполняемые строки (без учета комментариев, пустых строк, определения классов и типов).

Примером частично покрытых строк, может служить условный оператор со сложным предикатом, часть простых предикатов которого не покрыта тестами.

По результатам анализа исходного кода среда Visual Studio добавляет в редактор подсветку строк, отражающую степень покрытия кода. В среде Visual Studio используются следующие соглашения:

1) Светло-голубой цвет показывает, что строка была целиком выполнена в процессе тестирования.

2) Бежевый цвет показывает, что только часть блока/строки выполнена в процессе тестирования.

3) Красно-коричневый (Reddish Brown) цвет показывает, что строка не была в процессе тестирования.

2.3.3 Анализ покрытия кода с помощью AxoCover через Visual Studio

Среды разработки Visual Studio Professional и Community Editions (в отличие от Enterprise) не имеют встроенных средств для оценки покрытия кода (code/test coverage).

Однако эту возможность относительно легко можно добавить при помощи сторонних средств. Одним из популярных решений расширение AxoCover.

Рассмотрим пошаговую настройку работы Visual Studio 2015 для оценки покрытия тестов, написанных с помощью фреймворка NUnit.

Пусть имеется уже готовое решение (solution) с двумя проектами: некоторая библиотека STLab05 и проект с автоматизированными тестами. Добавим анализ покрытия кода для этого решения (рисунок 5.8).

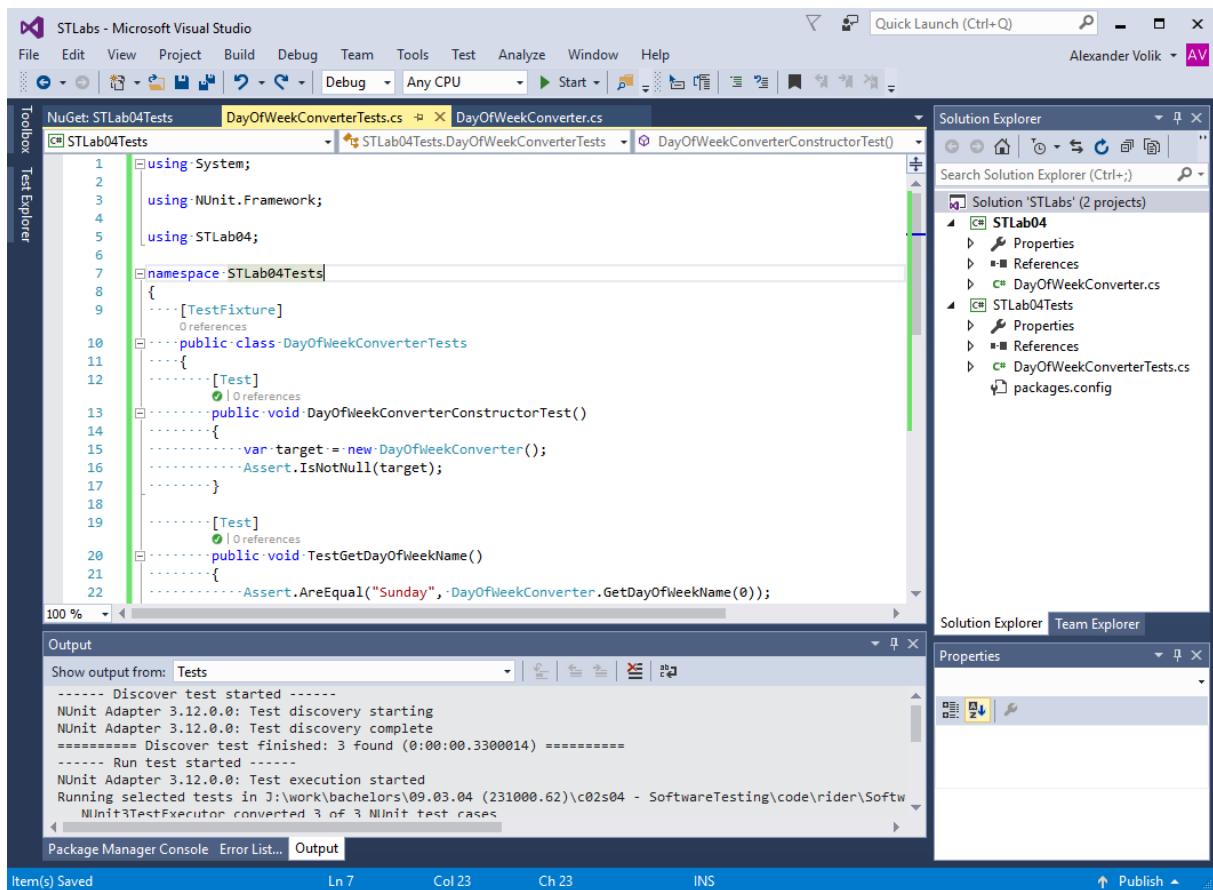


Рисунок 5.8 – Решение с автоматизированными тестами

Для начала необходимо установить расширение AxoCover для Visual Studio. Это можно сделать через пункт меню «Tools -> Extensions and Updates» (рисунок 5.9-5.10). Далее необходимо перезапустить Visual Studio.

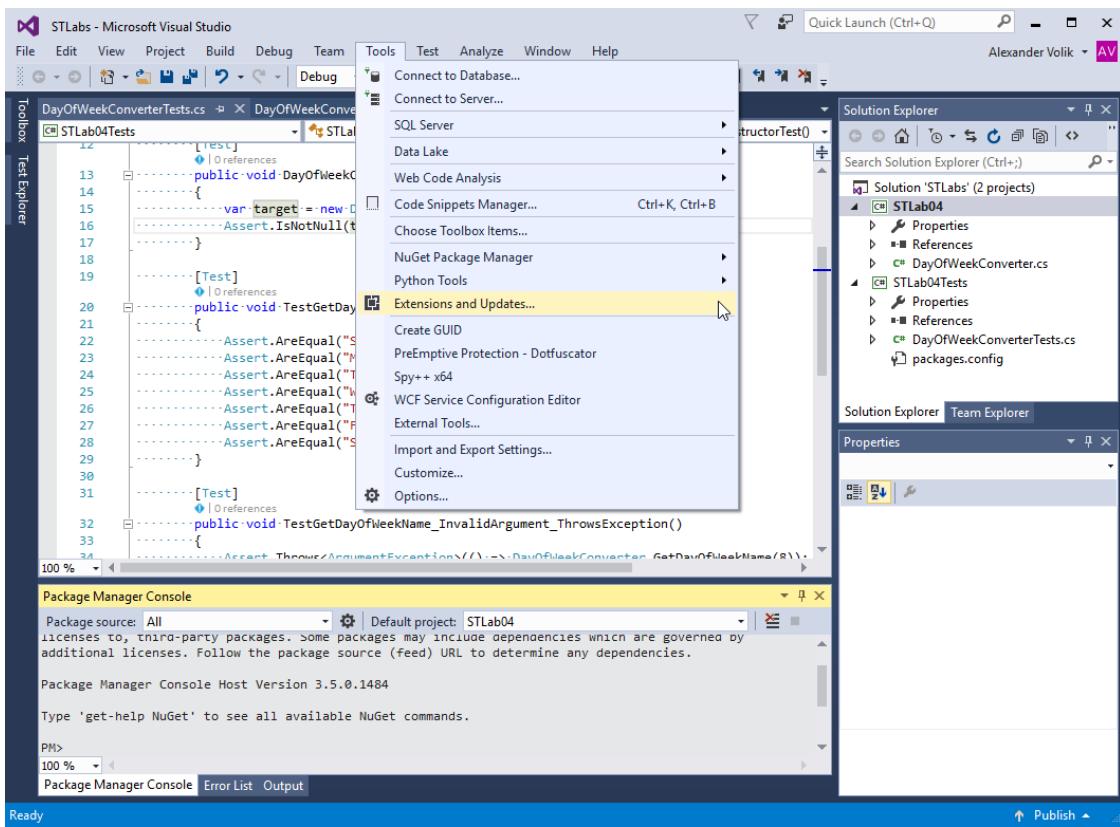


Рисунок 5.9 – Установка AxoCover

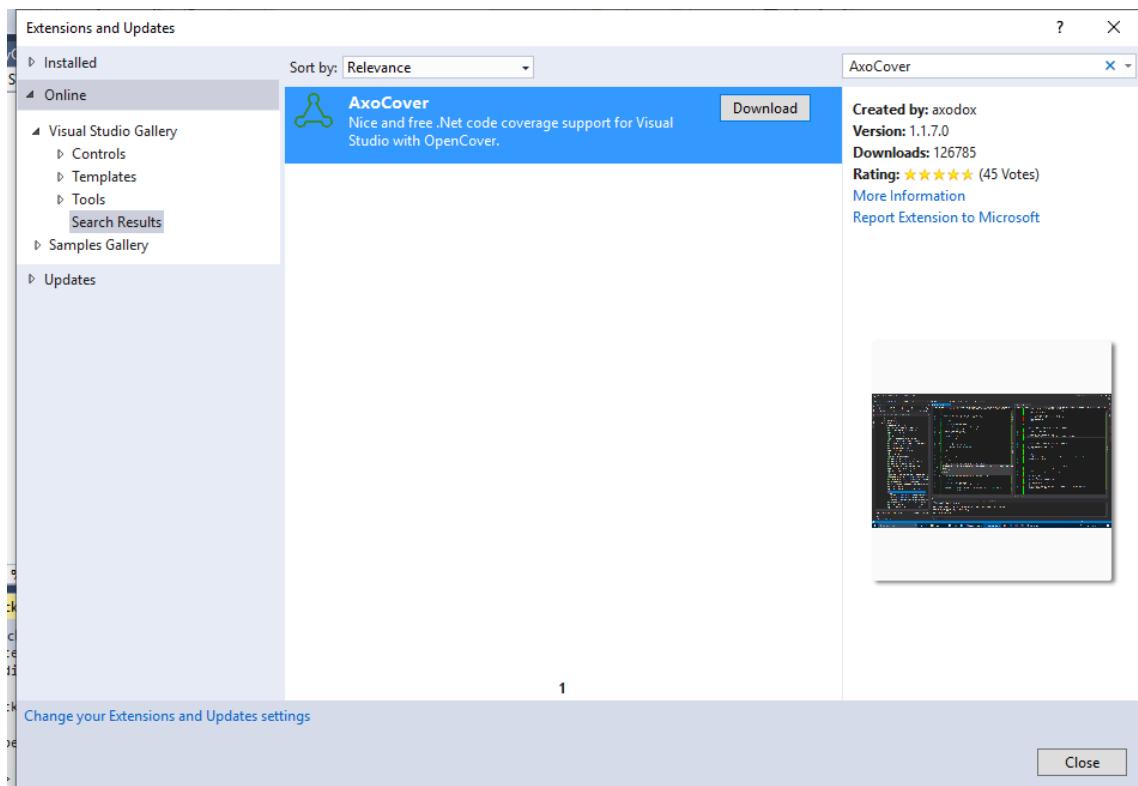


Рисунок 5.10 – Установка AxoCover

Далее необходимо открыть пункт «Tools -> AxoCover» для доступа к возможностям расширения (рисунки 5.11-5.12) и запустить анализ покрытия «Cover».

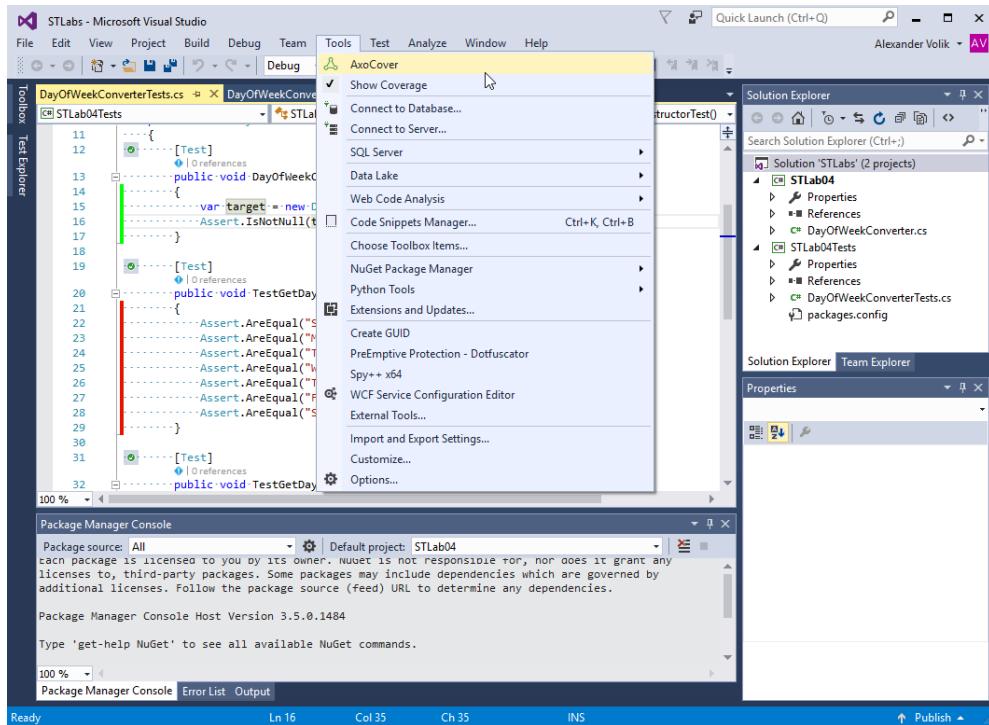


Рисунок 5.11 – Открытие AxoCover

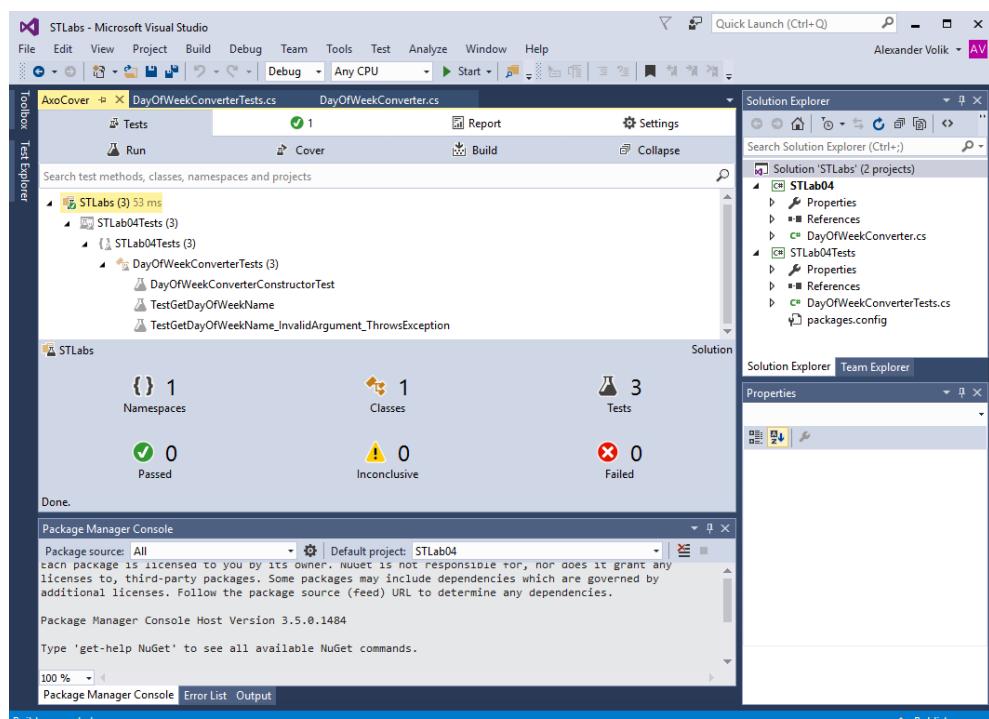


Рисунок 5.12 – Окно AxoCover

Результаты запуска тестов (рисунок 5.13) и покрытия можно увидеть на отдельной вкладке (рисунок 5.14).

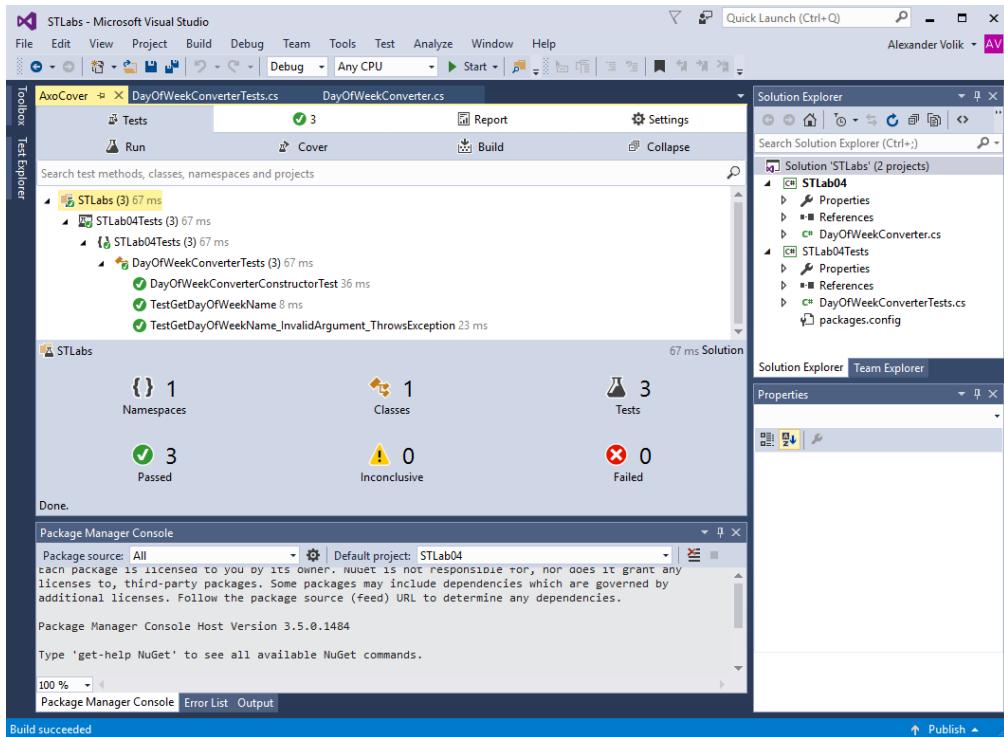


Рисунок 5.13 – Результаты запуска тестов в AxCover

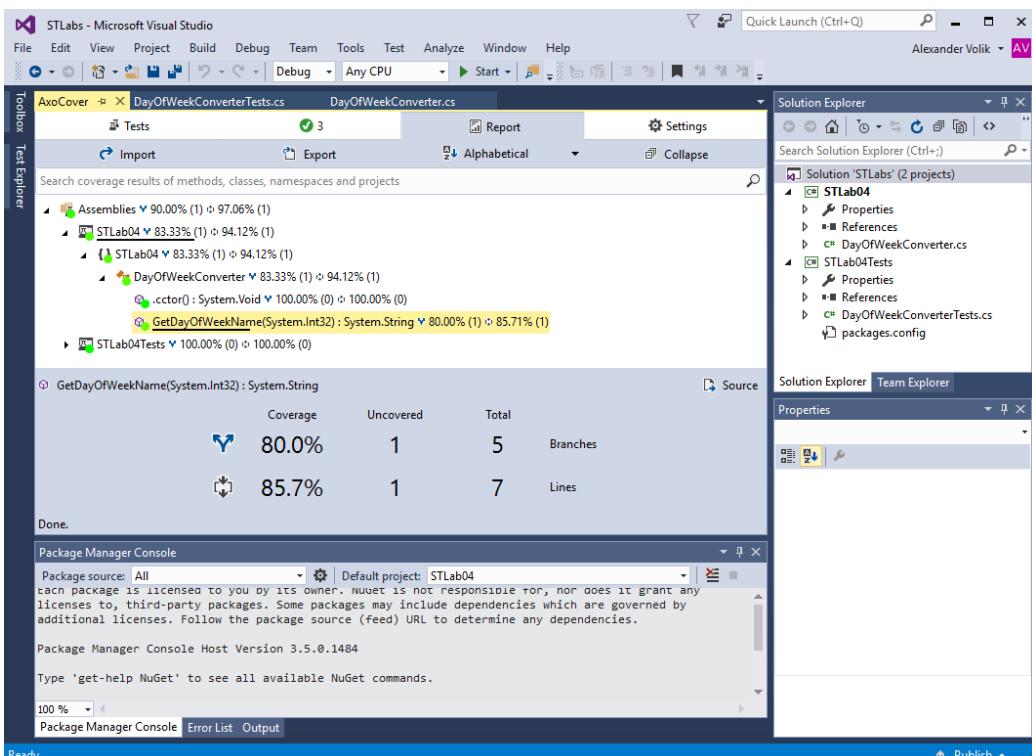


Рисунок 5.14 – Результаты анализа покрытия кода в AxCover

Также можно перейти к анализируемому коду и посмотреть его покрытие в среде Visual Studio (рисунок 5.15-5.16), а также экспортить результаты в формате HTML (пункт Export).

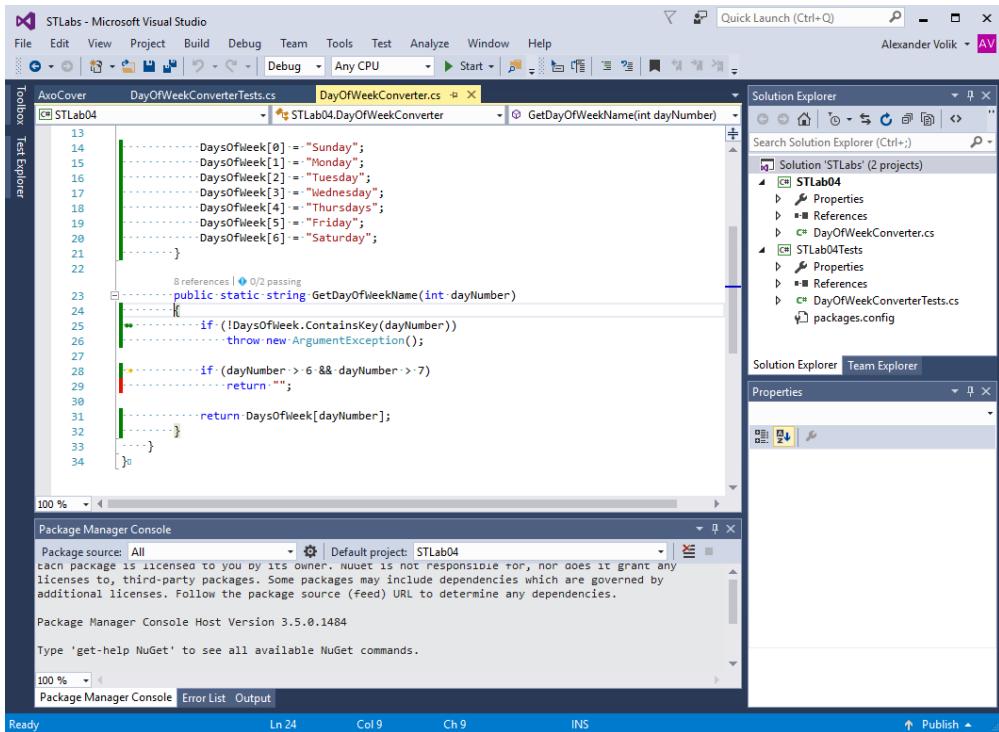


Рисунок 5.15 – Отображение покрытия кода в Visual Studio

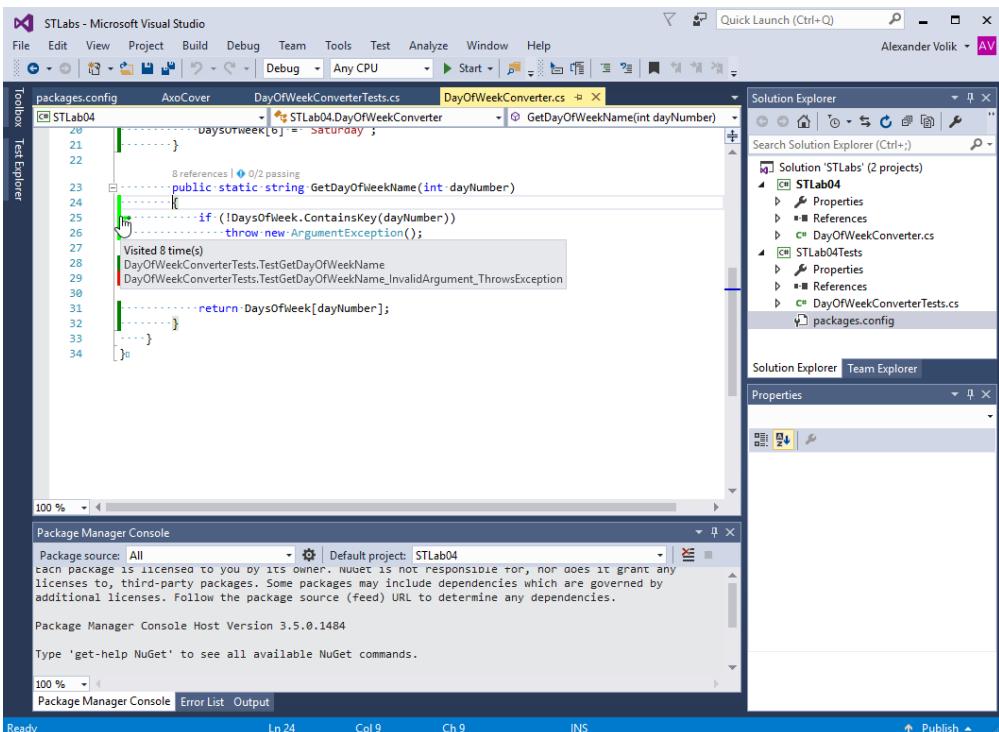


Рисунок 5.16 – Детали анализа покрытия строк кода в Visual Studio

В сгенерированном отчете отображается обобщенное значение покрытия классов и методов (рисунок 5.17), а в детальном отчете по каждому из методов (рисунок 5.18) цикломатическая сложность кода (Cyclomatic Complexity), покрытие операторов (Sequence Coverage) и ветвлений (Branch Coverage).

Также имеется возможность получения визуального представления информации о покрытии строк кода (рисунок 5.19).

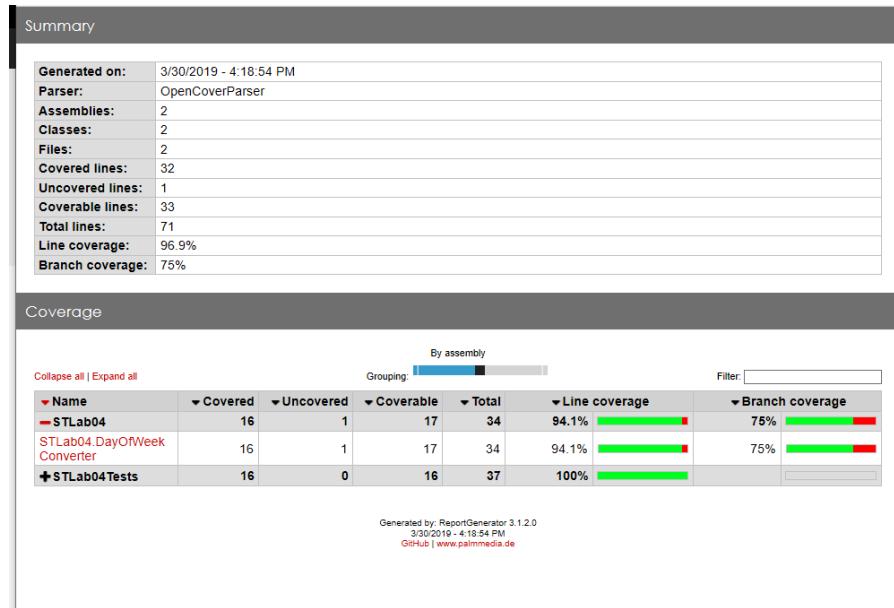


Рисунок 5.17 – Общий отчет анализа покрытия кода (OpenCover)

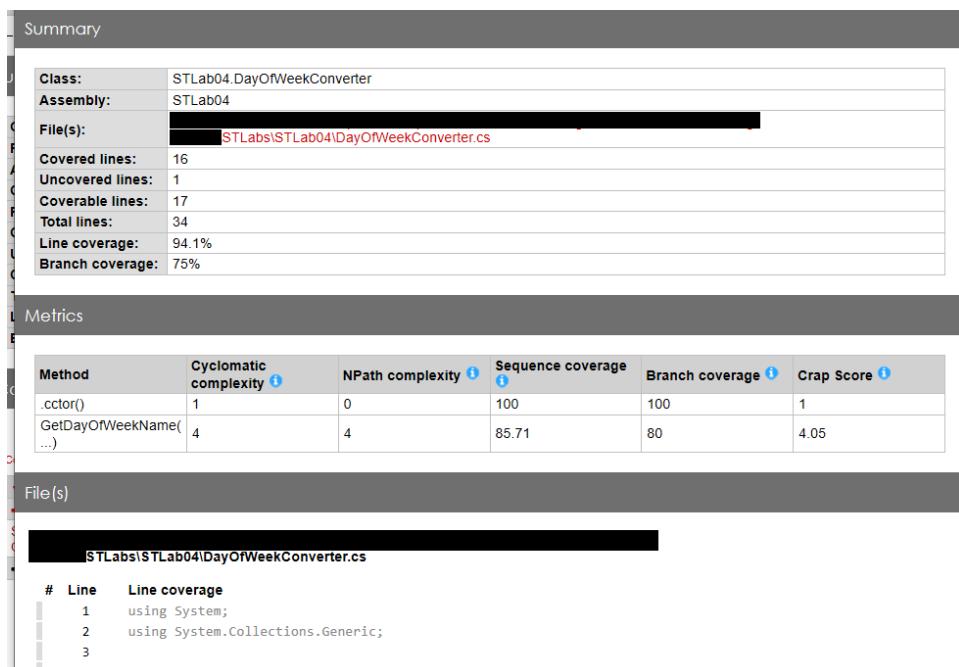


Рисунок 5.18 – Результаты анализа покрытия кода метода (OpenCover)

```

7     {
8         private static readonly Dictionary<int, string> DaysOfWeek;
9
10        static DayOfWeekConverter()
11        {
12            DaysOfWeek = new Dictionary<int, string>();
13
14            DaysOfWeek[0] = "Sunday";
15            DaysOfWeek[1] = "Monday";
16            DaysOfWeek[2] = "Tuesday";
17            DaysOfWeek[3] = "Wednesday";
18            DaysOfWeek[4] = "Thursday";
19            DaysOfWeek[5] = "Friday";
20            DaysOfWeek[6] = "Saturday";
21        }
22
23        public static string GetDayOfWeekName(int dayNumber)
24        {
25            if (!DaysOfWeek.ContainsKey(dayNumber))
26                throw new ArgumentException();
27
28            if (dayNumber > 6 && dayNumber > 7)
29                return "";
30
31            return DaysOfWeek[dayNumber];
32        }
33    }
34

```

Generated by: ReportGenerator 3.1.2.0
3/30/2019 - 4:18:54 PM
GitHub | www.palmmedia.de

Рисунок 5.20 – Покрытие кода (OpenCover)

2.3.4 Анализ покрытия кода с помощью OpenCover через консоль

Утилита OpenCover предназначена для анализа покрытия кода различными тестовыми фреймворками. Данная свободно распространяемая утилита с открытым исходным кодом может быть установлена через nuget-пакет. Помимо этого ее можно получить путем компиляции исходного кода из репозитория на GitHub (<https://github.com/sawilde/opencover>) или скачать в виде бинарной сборки с зеркала на BitBucket (<https://bitbucket.org/shaunwilde/opencover/downloads>).

Для получения удобочитаемого представления результатов можно воспользоваться утилитой ReportGenerator, позволяющей получить HTML представление этих данных.

Данная свободно распространяемая утилита с открытым исходным кодом может быть бесплатно получена сайта разработчиков <http://reportgenerator.codeplex.com/>. Там же имеется подробная документации по использованию приложения.

Рассмотрим запуск и работу с утилитой из консоли.

Для запуска необходимо передать в качестве параметров динамическую библиотеку (DLL) с тестами и фильтр для исключения из

отчета данных по работе самой библиотеки, а также имя выходного XML-файла с результатами анализа покрытия.

Фильтр представляет собой выражение для исключения или включения метода в отчет. В принятых в лабораторных работах соглашениях по именованию тестов (имя тестового пространства имен заканчивается на *Tests) достаточно использование фильтра "+[*]* - [*Tests]*" для исключения из отчета результатов работы самих тестовых методов.

Пример командного файла запуска такого теста представлен ниже (здесь выражение %1 используется для передачи имени тестируемого файла через параметры запуска командного файла и заменяется именем DLL с тестами).

Пример 5.4. Командный файл для запуска OpenCover с тестовым фреймворком MSTest

```
@echo off  
OpenCover.Console.exe -register:user -target:MSTest.exe -  
targetargs:"/testcontainer:%1" -filter:"+[*]* -[*Tests]*" -showunvisited  
-output:coverage.xml
```

Пример 5.5. Командный файл для запуска OpenCover с тестовым фреймворком NUnit

```
OpenCover.Console.exe -register:user -target:nunit-console-x86.exe -  
targetargs:"/noshadow SoftwareTestingLabsExamples05x01-ProjectTest.dll"  
-filter:+[*]* -output:coverage.xml
```

В результате мы получаем XML-файл (coverage.xml), однако данные, хранимые в нем не слишком удобны для анализа.

Для получения удобочитаемого представления результатов также воспользуемся утилитой ReportGenerator.

Для запуска необходимо передать в качестве параметра имя XML-файла с данными анализа и каталог для вывода результатов.

Пример 5.6. Командный файл для запуска ReportGenerator

```
ReportGenerator.exe "-reports:coverage.xml" "-targetdir:.\reports"
```

Сгенерированный отчет будет аналогичным тому, что был получен при экспорте результатов через AxoCover.

3 Контрольные вопросы

- 1) Опишите принцип тестирования, управляемого данными?
- 2) Что такое контекст тестирования?

- 3) Опишите принцип тестирования, управляемого данными?
- 4) Что такое «Параметризованное тестирование»?
- 5) Что такое покрытие кода?
- 6) По каким критериям можно оценивать покрытие кода?
- 7) Что такое частично выполненная строка?
- 8) Что такое цикломатическая сложность кода (Cyclomatic Complexity)?
- 9) В чем заключается критерий покрытия операторов?
- 10) В чем заключается критерий ветвлений (Branch Coverage)?

4 Задание

- 1) Используя класс из задания «Лабораторной работы №2. Тестирование методом черноного ящика», реализующий проверку принадлежности точки к области и тестовые наборы из той же работы создать проект тестирования (или добавить еще один тестовый класс в уже используемый тестовый проект) с набором тестов управляемых данными.
- 2) Запустить проект тестирования и проверить результаты работы.
- 3) Проанализировать покрытие кода тестами.
- 4) На основе тестов здания «Лабораторной работы №3. Тестирование методом белого ящика» проанализировать покрытие кода тестами.
- 5) Составить отчет о результатах проведенного тестирования.

Лабораторная работа № 6. Тестирование методом серого ящика

1 Цель работы

Изучить подход к тестированию методом серого ящика.

2 Краткая теория

Наиболее эффективная процедура тестирования заключается в том, чтобы разрабатывать тесты, используя стратегию черного ящика, а затем как необходимое условие – дополнительные тесты, используя методы белого ящика. Такой подход называют методом «серого ящика».

Это подход, сочетающий элементы двух предыдущих подходов. С одной стороны, тестирование, ориентированное на пользователя, а значит, мы используем паттерны поведения пользователя, т.е. применяем методику «Черного ящика». А с другой стороны производим «информированное» тестирование, т.е. использование знаний о том, как устроена хотя бы часть тестируемого кода.

Тестирование «черного ящика» может быть менее эффективным при поиске ошибок, связанных с потоком данных на уровне исходного кода. А тестирование «белого ящика» менее эффективно в нахождении высокоуровневых ошибок в операционной системе, а также ошибок совместимости. Тестирование же «серого ящика» используется для оценки проекта в рамках взаимодействия его индивидуальных компонентов.

Тестирование «серого ящика» наиболее подходит для тестирования веб-приложений, потому что для него необходимы высокоуровневая разработка, операционное окружение и условия совместимости.

Во время проведения анализа черного или белого ящика более сложно определить проблемы, связанные с непрерывным потоком данных. Специфические контекстные проблемы тестирования легче всего найти во время проверки серого ящика.

В большинстве случаев целесообразнее использовать подход «Серого ящика» так как он позволяет достигать преимущества обоих способов тестирования.

3 Контрольные вопросы

- 1) Какова цель тестирования?
- 2) Какие классы тестовых критериев вы знаете?
- 3) В чем заключается суть тестирования с помощью модели «черного ящика»?
- 4) Перечислите и охарактеризуйте функциональные критерии?
- 5) Какие стратегии тестирования методом черного ящика вы знаете?

- 6) В чем заключается суть тестирования с помощью модели «белого ящика»?
- 7) Перечислите и охарактеризуйте структурные критерии?
- 8) Какие стратегии тестирования методом белого ящика вы знаете?
- 9) В чем заключается суть тестирования с помощью модели «серого ящика»?

4 Задание

- 1) Создать класс (в соответствии с вариантом задания из п.5), реализующий преобразование строки из одной системы счисления в другую.
- 2) Протестировать класс на основе метода серого ящика с использованием средств автоматизации.
- 3) Составить отчет о результатах проведенного тестирования.

5 Варианты заданий

Реализовать класс, преобразующий строку с числом из одной системы счисления в другую. Варианты реализуемых преобразований представлены ниже (в ячейках таблицы указан номер варианта).

Исходное основание	Целевое основание					
	2	3	8	10	12	16
2	*	1	17	24	13	7
3	8	*	2	18	25	14
8	27	9	*	3	19	26
10	21	28	10	*	4	20
12	15	22	29	11	*	5
16	6	16	23	30	12	*

Лабораторная работа № 7. Модульное тестирование объектно-ориентированных программ

1 Цель работы

Изучить подход к автоматизации процесса модульного тестирования объектно-ориентированных программ.

2 Краткая теория

2.1 Модульное тестирование

Модульное тестирование – это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель модульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования.

Unit testing (юнит тестирование или модульное тестирование) — заключается в изолированной проверке каждого отдельного элемента путем запуска тестов в искусственной среде. Для этого необходимо использовать драйверы и заглушки. Поэлементное тестирование — первейшая возможность реализовать исходный код. Оценивая каждый элемент изолированно, и подтверждая корректность его работы, точно установить проблему значительно проще чем, если бы элемент был частью системы.

2.2 Цели модульного тестирования

Целью модульного тестирования является получение работоспособного кода с наименьшими затратами. И его применение оправдано тогда и только тогда, когда оно дает больший эффект, нежели другие методы.

Отсюда следует, что писать тесты для кода потенциально подверженного изменениям более выгодно, чем для кода, изменение которого не предполагается. Сложная логика меняется чаще, чем простая. Следовательно, в первую очередь имеет смысл писать модульные тесты на сложную логику. А на простую логику писать позднее или вообще тестировать другими методами.

Для того чтобы как можно реже изменять тесты следует хорошо планировать интерфейсы. То же самое можно сказать и применительно к написанию исходного кода. Действительно, создание хорошей архитектуры часто определяет дальнейший ход проекта. И есть оптимум, на каком этапе архитектура «достаточно хороша».

Если в проекте применяется модульное тестирование, то тщательное планирование интерфейсов становится более выгодным. Внедрению модульного тестирования должно предшествовать внедрение планирования интерфейсов.

Дополнительная работа по созданию тестов, их кодированию и проверке результатов вносит существенный вклад в общую стоимость проекта. И то, что продукт окажется более качественным не всегда перевешивает то, что он будет существенно дороже.

Известно, что продукт оптимальный по набору бюджет/функциональность/качество получается при применении различных способов обеспечения качества.

Если в результате исправления ошибок интеграции меняется исходный код, в нем с большой вероятностью появляются ошибки. Если в результате добавления новой функциональности меняется исходный код, в нем с большой вероятностью появляются ошибки. И искать их лучше с помощью ранее созданных модульных тестов.

2.3 Планирование тестов

При тестировании объектно-ориентированных программ можно использовать уже известные ранее подходы: методы чёрного и белого ящиков.

Метод белого ящика (White-box testing) для конструирования тестов использует внутреннюю структуру кода и управляющую логику. При этом

существует вероятность, что код будет проверяться так, как он был написан, а это не гарантирует корректность логики.

Метод черного ящика (Black-box testing) для конструирования тестов использует требования и спецификации программы. Недостатками являются невозможность найти взаимно уничтожающие ошибки, и редко возникающие ошибки (ошибки работы с памятью).

Основной проблемой тестирования является то, что хотя непротестированный код почти наверняка неработоспособен, но полное покрытие не гарантирует работоспособности. Написание тестов на основе уже существующего кода только для того, чтобы иметь стопроцентное покрытие кода тестами не дает никаких гарантий качества этого кода. Такой подход неизбежно приведет к существованию оттестированного, но неработоспособного кода. Кроме того, метод белового ящика, как правило, приводит к созданию позитивных тестов. А ошибки, как правило, находятся негативными тестами. В тестировании вопрос нахождения ошибки гораздо более важен и эффективен чем доказательство корректности. Поэтому в первую очередь тесты должны соответствовать не коду, а требованиям и должны базироваться на спецификации.

При подготовке тестового набора рекомендуется начать с простого позитивного теста. Затраты на его создание минимальны. Вероятность создания кода, не работающего в штатном режиме, гораздо меньше, чем отсутствие обработки исключительных ситуаций. Но исключительные условия в работе программы редки. Как правило, все работает в штатном режиме. Тесты на обработку некорректных условий, находят ошибки гораздо чаще, но если выяснится, что программа не обрабатывает штатные ситуации, то она просто никому не нужна.

Простой позитивный тест нужен т.к. несмотря на малую вероятность нахождения ошибки, цена пропущенной ошибки чрезмерно высока.

Последующие тесты должны создаваться при помощи формальных методик тестирования. Таких как, классы эквивалентности, исследование граничных условий и т.д. дополняя их тестами на основе знания особенностей реализации (структурой) программы.

Наиболее эффективный способ создания тестовых наборов – метод серого ящика, т.е. совместное использование методов черного и белого ящиков.

После этого следует проводить проверку полноты тестового набора с помощью формальной метрики «Code Coverage». Она показывает неполноту тестового набора. И дальнейшие тесты можно писать на основании анализа непротестированных участков.

2.4 Рассмотрение объекта как конечного автомата

В объектно-ориентированном программировании (ООП) объект класса можно рассматривать как автомат: состояние автомата — это состояние объекта; входной символ — операция в объекте с некоторым набором значений входных параметров; выходной символ — набор значений выходных параметров. Абстракция автомата применяется как для проектирования, так и для анализа проектных решений.

Класс в объектно-ориентированном программировании может быть рассмотрен как конечный автомат. Вызывая метод (компонент) некоторого класса, клиент указывает его имя и, если требуется, его фактические аргументы. Различных компонентов в классе обычно всего несколько, в то время как различных аргументов может быть необозримо много. При этом имя компонента определяет действие (алгоритм вычислений), а значения аргументов — только результат действия (результат вычислений).

Под состоянием класса обычно подразумевают множество текущих значений всех его полей. При этом текущее состояние полей определяются начальными значениями и историей воздействий на данный объект (порядком и аргументами вызовов методов). Поэтому зная историю и начальные занесения полей всегда можно определить текущее состояние объекта. Этот факт удобно использовать при тестировании объектно-ориентированных программ.

Для целей тестирования достаточно знать, что оракул автомата для любой допустимой пары <состояние, входной символ> может определить правильность перехода в новое состояние (правильность функции перехода) и правильность полученного выходного символа (правильность функции выхода). Задача тестирования объекта/класса как конечного автомата может решаться в предположении, что состояние конечного автомата доступно (в виде открытого поля или свойства) или не доступно непосредственному наблюдению со стороны тестовой системы (закрытое или защищенное поле/свойство). Если состояние не доступно (автомат как «чёрный ящик»), то тестовая система должна вести некоторое «абстрактное состояние», моделирующее реальное состояние автомата. В такой ситуации оракул вместо проверки функции перехода в новое реальное состояние вычисляет новое абстрактное состояние. Соответствие реального и абстрактного состояний устанавливается косвенным путём (через другие открытые поля/свойства) при проверках в оракуле функции выхода в процессе тестирования.

Так, например, при тестировании класса Stack, реализующего стек, мы непосредственно имеем доступ только к последнему добавленному элементу и к количеству элементов в стеке. И только по этим двум

значениям мы можем получать информацию о текущем внутреннем состоянии класса-контейнера. Т.е. для тестирования таких классов необходимо отслеживать внутреннее состояние класса через доступные поля и методы, делая по ним вывод о текущем внутреннем состоянии и ожидаемом поведении экземпляра класса.

Естественным критерием полноты тестового покрытия при тестировании автоматов является покрытие всех переходов автомата.

Для выполнения этого критерия необходимо генерировать все требуемые тестовые воздействия во всех состояниях автомата. Тем самым, задача генерации распадается на задачу «обхода» всех состояний автомата и задачу «перебора» тестовых воздействий для каждого состояния.

Одним из распространённых способов представления автомата является представление его в виде графа состояний (или графа переходов), вершины которого соответствуют состояниям автомата, а дуги — допустимым переходам. В терминах теории графов задача покрытия всех переходов автомата формулируется как задача обхода графа, то есть, прохождения по маршруту, содержащему все дуги графа.

Две основные проблемы, связанные с обходом графа состояний автомата: недетерминизм и слишком большой размер графа. Существует общий подход к решению обеих указанных выше проблем, основанный на введении классов эквивалентности вершин и дуг графа. Критерий покрытия всех дуг и вершин ослабляется до критерия покрытия всех классов эквивалентности.

Для тестирования объектно-ориентированных программ множество состояний и переходов объекта/класса удобно использовать UML диаграмму состояний (Statechart diagram).

2.5 Диаграммы состояний

В UML для моделирования поведения объекта с точки зрения порядка возникновения событий используются диаграммы состояний. Диаграммы состояний применяются для моделирования динамических аспектов системы. Имеется в виду обусловленное порядком возникновения событий поведение объектов любого рода в любом представлении системной архитектуры, включая классы, интерфейсы, компоненты и узлы.

Диаграмма состояний (Statechart diagram) показывает автомат, фокусируя внимание на потоке управления от состояния к состоянию.

Автомат (State machine) – это описание последовательности состояний, через которые проходит объект на протяжении своего жизненного цикла, реагируя на события, - в том числе описание реакций на эти события.

Состояние (State) – это ситуация в жизни объекта, на протяжении которой он удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого-то события.

Событие (Event) – это спецификация существенного факта, который происходит во времени и пространстве. В контексте автоматов событие – это стимул, способный вызвать срабатывание перехода.

Переход (Transition) – это отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, должен выполнить некоторые действия и перейти во второе состояние, как только произойдет определенное событие и будут выполнены заданные условия.

Деятельность (Activity) – это продолжающееся неатомарное вычисление внутри автомата.

Действие (Action) – это атомарное вычисление, которое приводит к смене состояния или возврату значения.

Диаграмма состояний изображается в виде графа с вершинами и ребрами. Пример диаграммы показан на рисунке 7.1.

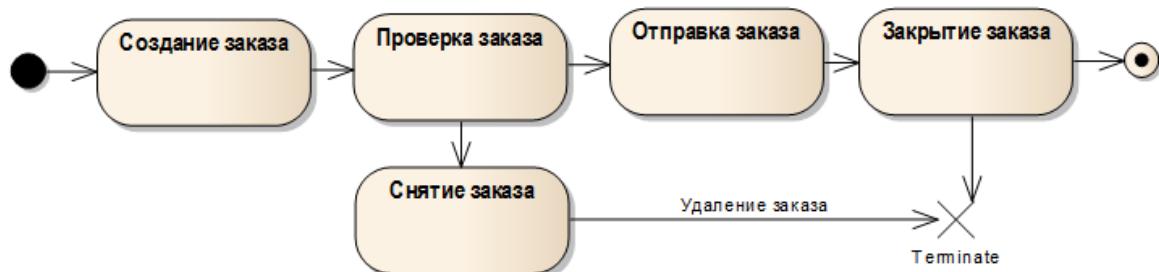


Рисунок 7.1 – Диаграмма состояний

Диаграмма состояний показывает все возможные состояния, в которых может находиться объект, а также процесс смены состояний в результате внешнего влияния.

Основными элементами диаграммы состояний являются «Состояние» и «Переход» (рисунок 7.2). Диаграмма состояний имеет схожую семантику с диаграммой деятельности, только деятельность здесь заменена состоянием, переходы символизируют действия. Таким образом, если для диаграммы деятельности отличие между понятиями «Деятельность» и «Действие» заключается в возможности дальнейшей декомпозиции, то на диаграмме состояний деятельность символизирует состояние, в котором объект находится продолжительное количество времени, в то время как действие моментально.

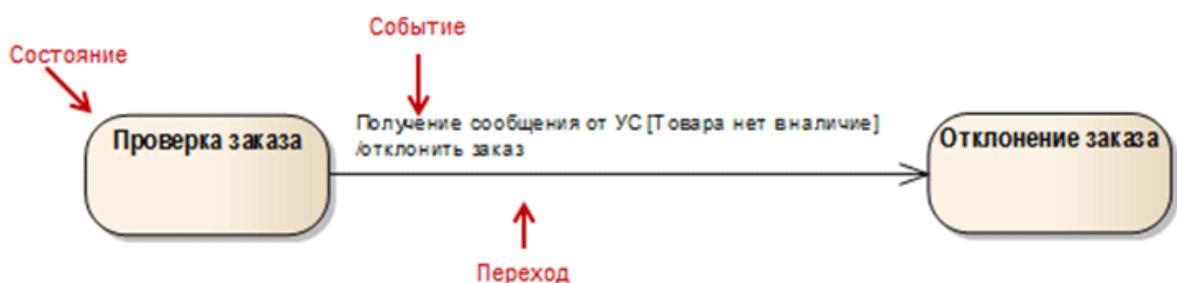


Рисунок 7.2 – Элементы диаграммы состояний

Переход может быть инициирован событием, которое также отражается на диаграмме состояний.

Состояние может содержать только имя или имя и дополнительно список внутренних действий. Список внутренних действий содержит перечень действий или деятельностей, которые выполняются во время нахождения объекта в данном состоянии.

Факт смены одного состояния другим изображается с помощью перехода. Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (do activity), получении объектом сообщения или приемом сигнала (подробнее события будут рассмотрены позднее). Переход может быть триггерным и нетриггерным. Если переход срабатывает, когда все операции исходного состояния завершены, он называется нетриггерным или переходом по завершении. Если переход инициируется каким-либо событием, он считается триггерным. Для триггерного перехода характерно наличие имени, которое может быть записано в следующем формате:

```
<имя события>'('<список параметров, разделенных запятыми>')'  
['<сторожевое условие>']' <выражение действия>.
```

Обязательным параметром является только имя события.

В качестве события могут выступать сигналы, вызовы, окончание фиксированных промежутков времени или моменты окончания выполнения определенных действий. После имени события могут следовать круглые скобки для явного задания параметров соответствующего события-триггера (например, пользователь инициирующий действие).

Помимо основных узлов, на диаграмме состояний могут использоваться, так называемые, псевдостатусы – вершины которые не обладают поведением, и объект не находится в ней, а «мгновенно» ее проходит.

Под псевдосостояниями на диаграмме состояний понимаются, знакомые уже нам начальное и конечное состояние. Начальное состояние обычно не содержит никаких внутренних действий и определяет точку, в которой находится объект по умолчанию в начальный момент времени. Конечное состояние также не содержит никаких внутренних действий и служит для указания на диаграмме области, в которой завершается процесс изменения состояний в контексте конечного автомата.

3 Контрольные вопросы

- 1) Какова цель модульного тестирования?
- 2) Опишите стратегии модульного тестирования?
- 3) Как осуществляется планирование тестов?
- 4) Как связаны объекты и конечные в автоматы?
- 5) Как конечные автоматы могут быть использованы при тестировании программ?
- 6) Что такое диаграмма состояний?
- 7) Какие компоненты диаграммы состояний вы знаете?

4 Задание

- 1) Создать класс (в соответствии с вариантом задания из п.5), реализующий работу с АСД (абстрактная структура данных).
- 2) Построить диаграмму состояний класса.
- 3) Составить тестовые требования к методам класса на основе полученной диаграммы состояний.
- 4) Определить наборы тестов на основе полученных тестовых требований.
- 5) Создать проект для автоматизированного модульного тестирования на основе тестовых наборов.
- 6) Запустить тестирование и проверить результаты работы.
- 7) Составить отчет о результатах проведенного тестирования.

5 Варианты заданий

	№ варианта														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Логическая структура (интерфейс)															
Стек	*					*					*				
Очередь		*					*					*			
Дек			*					*					*		
Список				*					*					*	
Множество					*					*					*
Физическое представление (реализация)															
Массив	*			*				*				*			*
Односвязный список		*			*	*			*				*		
Двусвязный список			*				*			*	*			*	

Лабораторная работа № 8. Тестирование с использованием тестовых двойников (Test Doubles)

1 Цель работы

Изучить подход к автоматизации процесса тестирования с использованием тестовых двойников (Test Doubles).

2 Краткая теория

Модульное тестирование представляет собой это тестирование программы на уровне отдельно взятых модулей, функций или классов. Однако между модулями и классами существуют взаимные зависимости. Когда один модуль или класс зависит от другого возникает проблема в тестировании зависимого класса. При возникновении ошибки в таких случаях не всегда можно сказать однозначно, кто является виновником проблемы: недостаточно протестированный класс нижнего уровня или сам тестируемый класс. Для исключения подобных эффектов необходимо иметь возможность однозначного определения источника проблемы.

В этом случае для успешного осуществления модульного тестирования необходимо эффективное замещение серверов компонентов в тестовых целях. Объекты-заместители обычно называются заглушками (stubs), или фиктивными объектами (mocks).

Термин «тестовый двойник», применяется как общее имя для объектов, используемых для замены настоящих компонентов в целях тестирования. Тестовый двойник является общим термином, а для конкретных реализаций используются специальные имена.

Различают пять типов тестовых двойников (таблица 8.1). Хотя в теории эти типы четко разделяются, разница размывается на практике.

Таблица 8.1 – Типы тестовых двойников

Тип тестового двойника	Описание
Объект-заглушка (Dummy Object)	Простейший, самый примитивный тип тестового двойника. Фиктивные модули не содержат реализации и в основном используются, когда необходимы только значения параметров. Пустые значения (null) могут считаться фиктивными модулями, но фиктивные модули как таковые являются производными от интерфейсов и базовых классов, не отягощенными реализацией.
Тестовая заглушка (Test Stub)	Тестовые заглушки (чаще употребляется просто термин заглушка) являются минимальными реализациями интерфейсов и базовых классов. Методы, возвращающие пустое значение (void), обычно не содержат реализаций вообще, тогда как методы, возвращающие значения, обычно возвращают жестко определенные значения.

Продолжение таблицы 8.1

Тестовый агент (Test Spy) (шпион)	Тестовый агент подобен заглушке, но помимо выдачи клиенту экземпляра, для вызова элементов, агент также записывает, какие элементы вызывались, так что модульные тесты могут удостоверяться, что элементы вызывались в соответствии с ожиданиями.
Поддельный объект (Fake)	Поддельный объект содержит более сложные реализации, обычно осуществляя взаимодействия между различными членами унаследованного ею типа. Поддельный объект не является полностью рабочим решением, но может напоминать таковое, хотя и с некоторыми упрощениями.
Фиктивный объект (Mock) (Имитация) / Подставной объект [Mez]	Фиктивный объект динамически создается библиотекой макетов (прочие типы обычно требуют работы с кодом от разработчика теста). Разработчик теста не видит реального кода, реализующего интерфейс, или базовый класс, но может настроить макет на предоставление возвращаемых данных, ожидание вызова определенных элементов и так далее. В зависимости от своей настройки макет может вести себя как фиктивный модуль, заглушка или агент.

2.1 Тестирование с помощью объектов-заглушек (dummy)

Объект-заглушка из-за своей примитивности гораздо проще в освоении, чем остальные типы двойников.

В качестве примера рассмотрим ситуацию сохранения набора данных в типизированный файл.

Рассмотрим простой класс DataManager, реализующий сохранение набора данных в типизированный файл (пример 8.1). Особое внимание стоит обратить на то, что конструктор принимает экземпляр ITypedFile.

Пример 8.1 Реализация класса DataManager

```
using System;
using System.Collections.Generic;

namespace STLab08
{
    public interface ITypedFile
    {
        string GetRecord(int id);
        void Save();
    }

    public class DataManager
    {
        private ITypedFile dataFile;
        private List<string> data;
        private int recordCount;

        public DataManager(ITypedFile dataFile)
        {
            this.dataFile = dataFile;
        }
    }
}
```

```

    {
        if (dataFile == null)
        {
            throw new ArgumentNullException("dataFile");
        }
        this.dataFile = dataFile;
        this.recordCount = 0;
        this.data = new List<string>();
    }

    public void AddData(string record)
    {
        if (record == null)
            throw new ArgumentNullException("record");
        this.recordCount++;
        this.data.Add(record);
    }

    public int RecordCount
    {
        get { return this.recordCount; }
    }

    public string GetFileRecord(int id)
    {
        if (id < 0)
            throw new ArgumentException("id");

        return dataFile.GetRecord(id);
    }

    public void Save()
    {
        this.dataFile.Save();
    }

    internal ITypedFile DataFile
    {
        get { return this.dataFile; }
    }
}
}

```

Хотя сам класс не используется непосредственно в процессе тестирования, но параметр конструктора не допускает передачу null.

Поэтому для модульного тестирования класса DataManager нужно создать тестовый двойник, реализующий ITypedFile, поскольку в противном случае конструктор выдаст исключение. Из тестовых

двойников простейшим решением в этом случае будет создание Dummy Object. При этом реализации всех элементов будут одинаковы (пусты).

После реализации класса DummyTypedFile, можно написать простой модульный тест для класса DataManager (пример 8.2).

Примечание. В данном примере класс DummyTypedFile расположен в том же модуле, что и тестирующий класс для упрощения процесса тестирования. В дальнейшем его можно выделить в отдельный модуль.

Пример 8.2 Реализация класса тестирования DataManager

```
using STLab08;
using NUnit.Framework;
using System;

namespace STLab08TestProject
{
    public class DummyTypedFile : ITypedFile
    {
        public string GetRecord(int id)
        {
            throw new Exception("The method not implemented.");
        }

        public void Save()
        {
            throw new Exception("The method not implemented.");
        }
    }

    [TestFixture()]
    public class DataManagerTest
    {
        [Test()]
        public void DataManagerConstructorTest()
        {
            ITypedFile dataFile = new DummyTypedFile();
            DataManager dataManager = new DataManager(dataFile);
            Assert.IsNotNull(dataManager);
        }

        [Test()]
        public void AddData_ValidRecord_IncreaseRecordCount()
        {
            ITypedFile dataFile = new DummyTypedFile();
            DataManager dataManager = new DataManager(dataFile);
            string record = string.Empty;
            dataManager.AddData(record);
        }
    }
}
```

```
        Assert.AreEqual(1, dataManager.RecordCount);
    }
}
}
```

На этом этапе достаточно Dummy Object, поскольку интерфейс ITypedFile не используется в методах, используемых на тестируемом объекте и просто необходимо преодолеть проверку входных данных в конструкторе. Очевидно, что как только будет вызван элемент, который реально использует ITypedFile (такой как метод Save), использование DummyTypedFile приведет к сбою теста, поскольку он выдаст исключение при вызове. Для таких тестов можно использовать иные типы тестовых двойников.

2.2 Тестирование с помощью заглушек (stub)

Как только модульный тест вызывает элемент тестируемого объекта, который, в свою очередь, вызывает элемент тестового двойника, то появляется необходимость, чтобы тест не выдавал при этом исключения. Простейшим типом двойника, отвечающего данному критерию, является заглушка (stub).

Примечание. Иногда возникает проблема точного определения отличий Объекта-заглушки (Dummy Object) от тестовой заглушки (Test Stub), но эта неопределенность никак не оказывается на практике, кроме как при выборе имен. В подавляющем большинстве случаев выгоднее реализовать полноценную заглушку, чем Dummy Object.

Например, при написании модульного теста,зывающего метод Save класса DataManager вовремя выполнения возникает исключение. Это происходит из-за того, что тестируемый метод вызовет метод Save интерфейса ITypedFile.

Соответственно потребуется реализовать метод Save таким образом, чтобы он не выдавал исключения (поскольку этот метод не возвращает значения, его реализация будет простой) (пример 8.3). Данная реализация заглушки может быть в дальнейшем использована и для других задач, например для тестирования метода GetFileRecord.

Пример 8.3 Реализация метода Save класса StubTypedFile

```
public class StubTypedFile : ITypedFile
{
    public string GetRecord(int id)
    {
```

```

        throw new Exception("The method not implemented.");
    }

    public void Save()
    {
        return;
    }
}

```

Имея класс StubTypedFile можно написать соответствующий модульный тест (пример 8.4).

Пример 8.4 Реализация тестирования Save

```

[Test()]
public void SaveTest()
{
    ITypedFile dataFile = new StubTypedFile();
    DataManager dataManager = new DataManager(dataFile);
    dataManager.Save();
    Assert.IsNotNull(dataManager);
}

```

Теперь рассмотрим тестирование метода и получающего запись из файла при помощи GetFileRecord. В этом случае нам необходимо будет усовершенствовать реализацию заглушки. Проще всего это сделать, возвратив жестко определенное значение (пример 8.5).

Пример 8.5 Простейшая реализация StubTypedFile

```

public class StubTypedFile : ITypedFile
{
    public string GetRecord(int id)
    {
        return "0";
    }

    public void Save()
    {
        return;
    }
}

```

Хотя предложенная реализация очень проста, она не отличается гибкостью, поскольку пригодна для тестирования только одного варианта работы. Использовать такую заглушки в других тестах будет очень проблематично. Одним из возможных решений является изменение реализации GetFileRecord путем добавления кода для анализа передаваемого параметра (пример 8.6).

Пример 8.6 Реализация StubTypedFile с анализом параметра

```
public class StubTypedFile : ITypedFile
{
    public string GetRecord(int id)
    {
        switch (id)
        {
            case 1:
                return "1";
            case 2:
                return "2";
            default:
                throw new ArgumentException("Unexpected id");
        }
    }

    public void Save()
    {
        return;
    }
}
```

В приведенном примере, в тестового двойника добавилась условная логика, делая его реализацию более сложной. В принципе, в метод можно добавить любое желаемое число операторов выбора, но, в этом случае простой просмотр тестового кода не дает ясного представления о том, какое значение возвратит заглушка.

Для решения этой проблемы введем в состав заглушки свойство ReturnedValue, содержащее возвращаемый результат, и изменим тело метода GetFileRecord так, чтобы он возвращал это значение (пример 8.7).

Приведенное решение делает данный тестовый двойник уже не заглушкой (stub), а поддельным объектом (fake).

Пример 8.7 Усовершенствованная реализация класса StubTypedFile

```
public class StubTypedFile : ITypedFile
{
    public string ReturnedValue
    {
        get;
        set;
    }

    public string GetRecord(int id)
    {
        return ReturnedValue;
    }
}
```

```

    public void Save()
    {
        return;
    }
}

```

В этом случае мы можем заставлять заглушку возвращать требуемый результат и спокойно проверять работу других частей программы (пример 8.8).

Пример 8.8 Реализация тестирования GetFileRecord

```

[TestMethod()]
public void GetFileRecordTest()
{
    string expected = string.Empty;

    StubTypedFile dataFile = new StubTypedFile();
    // Установим возвращаемое значение
    dataFile.ReturnedValue = expected;

    DataManager dataManager= new DataManager(dataFile);
    int id = 0;

    string actual;
    actual = dataManager.GetFileRecord(id);
    Assert.AreEqual(expected, actual);
}

```

Такая реализация вызывает `GetRecord` у `ITypedFile`. Однако она никак не реагирует на передаваемое значение `id` внутри своей логики.

2.3 Тестирование с помощью агентов (spy)

Целью модульного теста является демонстрация выполнения тестируемым объектом возложенных на него задач. Задачей метода `Save` является вызов метода `Save` на интерфейсе `ITypedFile`, но тест при помощи заглушки (пример 8.4) не проверяет этого, поскольку он завершится успешно, даже если реализация `dataManager.Save()` пуста (если не произойдет вызова `ITypedFile.Save`).

Чтобы проверить, произошло ли обращение к методу `Save`, тестовый двойник должен записать, был ли он вызван, или нет. Записи вызовов элементов для последующей проверки являются основной отличительной чертой тестового агента.

Создадим новый класс тестового агента SpyTypedFile (пример 8.9) и изменим тест метода Save (пример 8.10).

Пример 8.9 Реализация класса SpyTypedFile

```
public class SpyTypedFile : ITypedFile
{
    public SpyTypedFile()
    {
        this.SaveWasInvoked = false;
    }

    public bool SaveWasInvoked
    {
        get; set;
    }

    public string GetRecord(int id)
    {
        throw new Exception("The method not implemented.");
    }

    public void Save()
    {
        SaveWasInvoked = true;
        return;
    }
}
```

В этом случае тест будет выглядеть следующим образом (пример 810).

Пример 8.10 Реализация тестирования Save

```
[Test()]
public void SaveTest_with_Confirmation()
{
    SpyTypedFile dataFile = new SpyTypedFile();
    DataManager dataManager = new DataManager(dataFile);
    dataManager.Save();
    Assert.IsTrue(dataFile.SaveWasInvoked);
}
```

Этот простой тест проверяет лишь факт вызова метода Save, но не передаваемые параметры. Зачастую стоит проверять и подобные данные, но для этого придется либо написать более совершенный тестовый агент или обратиться к макетам, которые обычно установлены на запись всех вызовов по умолчанию.

Запись вызовов элементов можно сделать разными способами. В предыдущем примере был использован флаг, но вместо этого можно было использовать запись количества вызовов (пример 8.11).

Пример 8.11 Альтернативная реализация класса-агента

```
public class SpyCountTypedFile : ITypedFile
{
    public SpyCountTypedFile()
    {
        this.SaveWasInvoked = 0;
    }

    public int SaveWasInvoked
    {
        get;
        set;
    }

    public string GetRecord(int id)
    {
        throw new Exception("The method not implemented.");
    }

    public void Save()
    {
        SaveWasInvoked++;
        return;
    }
}
```

В этом случае тест будет выглядеть следующим образом (пример 8.12).

Пример 8.12 Альтернативная реализация тестирования Save

```
[Test()]
public void SaveTest_InvokeSave_InvokeCountEqualsOne()
{
    SpyCountTypedFile dataFile = new SpyCountTypedFile();
    DataManager dataManager = new DataManager(dataFile);
    dataManager.Save();
    Assert.AreEqual(1, dataFile.InvokeCount);
}
```

Такая реализация явно собирает больше информации, чем предыдущая. Создание еще более совершенных агентов возможно путем записи параметров ввода для каждого вызова, но это требует больше кода для реализации и, следовательно, приблизит его к fake- и mock-объектам.

2.4 Тестирование с помощью поддельных объектов (fake)

Создание более открытого для настройки тестового двойника может быть удобным в некоторых случаях. В случае ITypedFile данный интерфейс является просто абстракцией базы данных, так что одним из подходов может быть создание примитивной, хранящейся в памяти базы данных. Класс FakeTypedFile, прост настолько, насколько это возможно. Он содержит объект коллекции, который, в сущности, служит заменителем реального типизированного файла (пример 8.13). Поскольку для тестирования необходима возможность получать данные по индексу, достаточно создать словарь значений, фактически являющийся набором тестовых данных.

Пример 8.13 Реализация fake-объекта

```
public class FakeTypedFile : ITypedFile
{
    public Dictionary<int, string> values;

    public FakeTypedFile()
    {
        this.values = new Dictionary<int, string>();
    }

    public string GetRecord(int id)
    {
        if (this.values.ContainsKey(id))
            return values[id];
        throw new Exception("The method not implemented.");
    }

    public void Save()
    {
        return;
    }
}
```

FakeTypedFile не сохраняет значения, при вызове метода Save.

Для простоты, рассматриваемые в лабораторной тесты не проверяют элементы, добавленные при работе и сохранённые потом. В противном случае мы не могли бы получить вновь добавленные и сохранённые элементы. Однако если бы в этом возникла необходимость, то можно было бы модифицировать внутренний словарь и затем проверять его содержимое. Это добавило бы fake-объекту свойство тестового агента.

Класс FakeTypedFile позволяет теперь контролировать тестовые данные. Прежде чем создавать тестируемый объект, создается fake-объект

и заполнится список требуемых значений. При выполнении теста GetRecord возвращает значения для запрашиваемого индекса в списке.

Пример 8.14 Тестирование GetFileRecord с помощью fake-объекта

```
[Test()]
public void GetFileRecordFakeTest()
{
    FakeTypedFile dataFile = new FakeTypedFile();
    // Заполняем тестовый набор данных
    dataFile.values.Add(0, "0");
    dataFile.values.Add(1, "1");
    dataFile.values.Add(2, "2");
    dataFile.values.Add(4, "4");

    string expected = "1";

    DataManager dataManager = new DataManager(dataFile);
    int id = 1;

    string actual;
    actual = dataManager.GetFileRecord(id);
    Assert.AreEqual(expected, actual);
}
```

Можно заметить, что этот тест теперь легче читать, чем подобный тест, использующий заглушку, поскольку не нужно переключаться на ее код, чтобы посмотреть, какие значения он возвратит. Недостаток fake-объекта заключается в том, что его создание само по себе может потребовать некоторой работы.

2.5 Тестирование с помощью фиктивных объектов (mock)

Одно из полезных правил модульного тестирования заключается в том, чтобы тестировать сложный код при помощи относительно простого кода. Создание сложных дублеров идет с ним несколько вразрез. И если возникает необходимость тестировать самого дублера, то следует отказаться от такого решения.

Динамические мок-объекты (DynamicMock), создаваемые при помощи специальных фреймворков, представляют собой идеальное решение этой проблемы. Но использование полной библиотеки мок-объектов не всегда является верным выбором. Библиотека мок-объектов – это лишняя сборка, на которую придется ссылаться в тестовом проекте; а при работе над коллективным проектом решать вопросы стандартизации общей библиотеки мок-объект, установки ее на всех машинах разработчиков (или помещения двоичного файла под управление исходными файлами) и прочие проблемы, связанные с развертыванием

приложений. По этой причине порой предпочтительней воспользоваться создаваемым вручную mock-объект (manual mock).

Положим, нужно переписать тест примера 8.14 не используя fake-объект, но оставляя работу тестового двойника прозрачной.

Единственным элементом ITypedFile, влияющим на тест, является метод GetFileRecord. Поэтому создадим настраиваемый вручную mock-класс для работы с этим конкретным методом (пример 8.15). Конструктор данного класса принимает делегат Converter<int, string> и класс использует его в дальнейшем в реализации метода GetFileRecord.

Пример 8.15 Реализация создаваемого вручную макета

```
public class ManualMockTypedFile : ITypedFile
{
    private Converter<int, string> conv;

    public ManualMockTypedFile(Converter<int, string> convCallback)
    {
        this.conv = convCallback;
    }

    public string GetRecord(int id)
    {
        return this.conv(id);
    }

    public void Save()
    {
        throw new Exception("The method not implemented.");
    }
}
```

Converter<int, string> – делегат, принимающий целое как входящие данные и возвращающий строку. Если обратиться к определению ITypedFile, то можно заметить, что такая же сигнатура у метода GetFileRecord. Это означает, что можно использовать Converter<int, string> для реализации GetFileRecord.

Создаваемый вручную макет неприменим сам по себе, а используется для модульного теста (пример 7.16). В данном модульном тесте mock-объект инициализируется с помощью анонимного метода, который будет вызван при обращении к методу GetFileRecord. По сути, это подобно настройке динамических mock-объектов, когда сначала определяются ожидания для mock-объекта, а затем создается и используется тестовый объект.

Пример 8.16 Тестирование GetFileRecord с помощью mock-объекта

```
[Test()]
public void GetFileRecordManualMockTest()
{
    ManualMockTypedFile dataFile = new ManualMockTypedFile(
        delegate(int id)
    {
        switch (id)
        {
            case 1:
                return "1";
            case 2:
                return "2";
            default:
                throw new ArgumentOutOfRangeException("id");
        }
    });
    string expected = "1";

    DataManager dataManager = new DataManager(dataFile);
    int idx = 1;

    string actual;
    actual = dataManager.GetFileRecord(idx);
    Assert.AreEqual(expected, actual);
}
```

Хотя приведенный модульный тест может выглядеть более сложным, чем тест с использованием fake-объекта (пример 8.14), это не всегда так. Хотя сам тест и более развернут, общий объем кода, поддерживающего тест, меньше, поскольку fake-объект требует поддерживающий класс Dictionary (который, в более сложных случаях, потребует создание дополнительного класса для хранения данных), тогда как создаваемый вручную mock-объект требует только сам себя.

Достоинство использования делегатов при работе с mock-объектами заключается в том, что разработчик теста пишет новую реализацию для каждого модульного теста. Это позволяет не только менять возвращаемые значения и поведение, но и определять уровень выполняемой слежки. Если достаточно возвращения значений, можно обойтись кодом, подобным примеру 8.14, но если необходимо еще и записать вызовы методов для последующей проверки, это также легко сделать, поскольку анонимные методы позволяют доступ к внешним переменным.

Основной недостаток класса ManualMockTypedFile заключается в том, что он создает лишь mock-объект метода GetFileRecord, но выдает

NotImplementedException в другом элементе. Очевидно, что концепцию создаваемых вручную мок-объектов можно серьезно расширить, сделав подход более общим. В созданном вручную макете общего назначения для ITypedFile можно определить Converter<TInput, TOutput> для каждого элемента интерфейса, но в интерфейсе с множеством элементов такой ход скоро станет неудобным.

Преимуществом мок-объектов, созданных с помощью делегатов, заключается в высокой степени свободы, предлагаемой ими и сравнительно низкой сложности освоения.

2.6 Сравнение типов тестовых двойников

В таблице 8.2 перечислены различные типы тестовых двойников, их достоинства и недостатки.

Таблица 8.2 – Сравнение типов тестовых двойников

Тип тестового двойника	Достоинства	Недостатки
Объект-заглушка (Dummy Object)	Простота создания	Низкая эффективность использования.
Тестовая заглушка (Test Stub)	Простота создания	Негибкость. Непрозрачность для наблюдения из модульного теста. Отсутствие возможности удостовериться в верном вызове элементов.
Тестовый агент (Test Spy)	Возможность удостовериться в верном вызове элементов.	Негибкость. Непрозрачность для наблюдения из модульного теста.
Поддельный объект (Fake)	Предоставление полуготовой реализации, которая может быть использована во многих различных ситуациях.	Сложность создания. Реализация может быть достаточно сложной, чтобы требовать модульного тестирования уже для самого тестового дублера
Фиктивный объект (Mock)	Эффективное создание тестовых двойников. Возможность удостовериться в верном вызове элементов. Прозрачность для наблюдения из модульного теста.	Сложность в реализации и практическом применении

2.7 Разделение по проверке тестовых результатов

Поскольку четкой границы между типами двойников нет, и большая часть из них является взаимозаменяемой, то с точки зрения тестирования

их можно на две категории: предоставляющие данные (условно можно назвать их заглушками, stubs) и фиксирующие вызовы (условно можно назвать их mock-объектами).

В первом случае проверку правильности выполнения осуществляют над результатами работы самого тестируемого объекта.

Во втором случае проверку правильности выполнения осуществляют уже над результатами работы тестового двойника.

Таким образом, если необходимо предоставить тестируемому коду определенный набор данных (возможно даже с учетом входных данных), то следуют использовать заглушки.

Если же необходимо получить подтверждение вызова метода некоторого метода из тестируемого кода, проверить правильность передаваемых при этом параметров, количество вызовов и т.д., то следует воспользоваться mock-объектами.

2.8 Инверсия управления и внедрение зависимостей

При разработке программного обеспечения желательно разделить процессы загрузки/выгрузки информации и процесс непосредственной обработки полученных данных. Удобно иметь возможность произвольно менять форматы хранения данных в файлах, получая при этом одно и то же представление информации для обработки. Для реализации такой возможности удобно использовать внедрение зависимостей.

Инверсия управления (Inversion of Control, IoC) – принцип объектно-ориентированного программирования, используемый для уменьшения зацепления (связанности между компонентами).

Если объекту нужно получить доступ к определенному сервису (например, сохранение в файл), он получает ссылку на этот сервис и использует ее для решения своих задач. При этом процесс передачи ссылки можно контролировать извне.

Такое решение позволяет изменять поведение уже существующего объекта во время выполнения программы и гибко настраивать работу его без перекомпиляции.

Одним из вариантов реализации инверсии управления является внедрение зависимостей.

Внедрение зависимости (Dependency injection, DI) – процесс предоставления внешней зависимости программному компоненту.

Можно выделить три основных стиля внедрения зависимости под следующими названиями: Constructor Injection, Setter Injection, и Interface Injection. В данном примере будут реализованы первые два варианта

Пускай имеется некоторый компонент, который умеет возвращать список фильмов конкретного режиссера.

Эту функциональность можно реализовать одним простым методом:

```
class MovieLister
{
    public Movie[] MoviesDirectedBy(string director)
    {
        List<Movie> result = new List<Movie>();

        List <Movie> allMovies = Finder.FindAll();
        foreach(var movie in allMovies)
        {
            if (movie.Director == director)
            {
                result.Add(movie);
            }
        }

        return result.ToArray();
    }
}
```

Реализация этой функции примитивна, она просит объект поиска (до которого мы доберемся через мгновение) вернуть список всех известных ему фильмов. Затем выбирает из этого списка фильмы, принадлежащие конкретному режиссеру.

Удобно, чтобы метод MoviesDirectedBy был полностью независим от того, как этот список хранится (в файле, базе данных или обычном массиве). Этот метод просто ссылается на объект поиска (Finder), который умеет возвращать все фильмы, а затем обрабатывает полученный список.

В качестве одного из решений можно вынести эту зависимость из метода путем определения интерфейса для поиска.

```
interface MovieFinder
{
    List<Movie> FindAll();
}
```

Теперь код поиска и обработки разнесен, и когда потребуется реальный класс, который знает, как на самом деле хранятся фильмы, его можно будет, например, создать в конструкторе:

```
class MovieLister
{
    private MovieFinder Finder;

    public MovieLister()
    {
```

```

        Finder = new CSVFinder("movies1.txt");
    }
}

```

Подразумевается, что класс реализации получает список из текстового файла с разделителями (CSV-файл).

Если использовать эту реализацию класса MovieLister только для работы с одним типом файлов, то все будет хорошо, однако если потребуется одновременная работа с несколькими видами источников, то такое решение будет негибким и потребует постоянного изменения кода.

Хотя полученный и отделяет получения списка фильмов от их обработки, он по-прежнему не очень гибок. Изменить эту ситуацию возможно, если конкретный экземпляр класса поиска, можно будет указать, например, в конструкторе или задать через свойство:

```

class MovieLister
{
    public MovieFinder Finder { get; set; }

    public MovieLister()
    {
        Finder = new CSVFinder("movies1.txt");
    }
}

```

2.9 Использование внедрение зависимостей при тестировании

Теперь рассмотрим следующий пример, позволяющий понять, как использование внедрения зависимостей упрощает процесс тестирования.

Пускай в файле хранится список ребер, который потом необходимо преобразовать в матрицу смежности, для обхода графа в ширину. При этом необходимо иметь возможность экспорттировать граф в виде списка связности и сохранить его в файл.

Таким образом, получается седеющая цепочка преобразований данных:

- Файл со списком ребер (данные в файле)
- => Список ребер (как структура данных)
- => Матрица смежности (полученная из списка ребер и используемая для обхода графа)
- => Матрица инцидентности (как структура данных, полученная из матрицы смежности)
- => Файл с матрицей инцидентности (представление данных в файле)

С точки зрения указанных преобразований удобно разделить процессы загрузки/выгрузки графа и процесс обработки самого графа

(реализации алгоритмов и преобразований форматов представления графов). Данное разделение позволяет иметь возможность произвольно менять форматы хранения данных в файлах, получая при этом одно и то же представление графа, которое потом можно преобразовывать и использовать для работы.

Для этого необходимо выделить 2 интерфейса для организации чтения и записи, которые нужно будет использовать в классе для работы с графиками (их можно внедрить через конструктор или свойство). А также создать класс или классы, реализующие эти интерфейсы. Пример схемы взаимодействия таких классов представлен на рисунке 8.1.

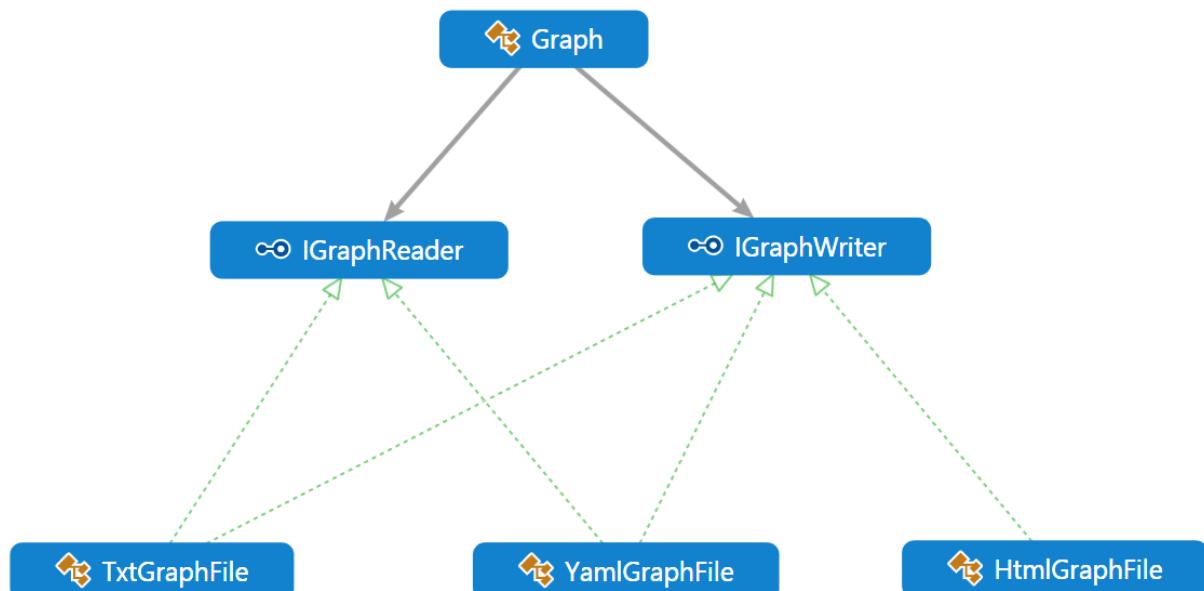


Рисунок 8.1 – Диаграмма взаимодействия классов

Использование подобной схемы позволяет гибко настраивать работу класса для работы с графиками, а также упрощает процесс тестирования. При тестировании можно создать экземпляр класса IGraphReader, который не будет работать с реальными файлами, а будет возвращать заранее известные значения, то есть, по сути, будет представлять собой тестовую заглушку.

Примеры реализации такой схемы приведены ниже.

Пример 8.16 Класс Graph

```
using System;

namespace STLab08Example.Variant1
{
    class Graph
```

```

// Setter Injection
public IGraphReader Reader { get; set; }

// Setter Injection
public IGraphWriter Writer { get; set; }

int[,] AdjacencyMatrix;

// Constructor Injection
public Graph(IGraphReader reader, IGraphWriter writer)
{
    this.Reader = reader;
    this.Writer = writer;
    AdjacencyMatrix = null;
}

public void LoadFromFile(string path)
{
    List<Edge> edges = Reader.LoadEdgeList(path);

    //Convert to AdjacencyMatrix
    AdjacencyMatrix = new int[0, 0];
}

public void SaveToFile(string path)
{
    //Convert form AdjacencyMatrix to IncidenceMatrix
    int[,] incidenceMatrix = new int[0, 0];

    Writer.SaveIncidenceMatrix(path, incidenceMatrix);
}

public int[] BFS()
{
    int[] traversePath = new int[0];
    //

    return traversePath;
}
}
}

```

Пример 8.17 Интерфейсы IGraphReader и IGraphWriter

```

using System;

namespace STLab08Example.Variant1
{

interface IGraphReader
{
    List<Edge> LoadEdgeList(string path);
}

```

```

    }

    interface IGraphWriter
    {
        void SaveIncidenceMatrix(string path, int[,] incidenceMatrix);
    }
}

```

Пример 8.18 Прототипы реализации IGraphReader и IGraphWriter в виде классов TxtGraphFile, HtmlGraphWriter и YamlGraphFile

```

using System;

namespace STLab08Example.Variant1
{
    class TxtGraphFile : IGraphReader, IGraphWriter
    {
        public List<Edge> LoadEdgeList(string path)
        {
            throw new NotImplementedException();
        }

        public void SaveIncidenceMatrix(string path,
                                         int[,] incidenceMatrix)
        {
            throw new NotImplementedException();
        }
    }

    class HtmlGraphWriter : IGraphWriter
    {
        public void SaveIncidenceMatrix(string path,
                                         int[,] incidenceMatrix)
        {
            throw new NotImplementedException();
        }
    }

    class YamlGraphFile : IGraphReader, IGraphWriter
    {
        public List<Edge> LoadEdgeList(string path)
        {
            throw new NotImplementedException();
        }

        public void SaveIncidenceMatrix(string path,
                                         int[,] incidenceMatrix)
        {
            throw new NotImplementedException();
        }
    }
}

```

```
}
```

Пример 8.19 Пример использования класса Graph

```
using System;

namespace STLab08Example.Variant1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Set default reader and writer
            IGraphReader r = new TxtGraphFile();
            IGraphWriter w = new YamlGraphFile();

            // Create Grpah object
            Graph graph = new Graph(r, w);

            // Load graph from TXT file
            string pathTxt = "EdgeListGraph.txt";
            graph.LoadFromFile(pathTxt);

            // Start BFS
            int[] traversePath = graph.BFS();

            // Save to YAML
            string pathYaml = "IncidenceMatrixGraph.yaml";
            graph.SaveToFile(pathYaml);

            // Change output formater to HTML
            graph.Writer = new HtmlGraphWriter();

            // Save to HTML
            string pathHtml = "IncidenceMatrixGraph.html";
            graph.SaveToFile(pathHtml);
        }
    }
}
```

Пример 8.20 Пример тестовой заглушки и тестового агента (реализующих интерфейсы IGraphReader и IGraphWriter)

```
using System;
using System.Collections.Generic;
```

```

namespace STLab08Example.Variant1
{
    class ReaderStub : IGraphReader
    {
        public List<Edge> LoadEdgeList(string path)
        {
            return new List<Edge>()
            {
                new Edge() {Start = 1, End = 2 },
                new Edge() {Start = 2, End = 3 },
                new Edge() {Start = 2, End = 5 },
                new Edge() {Start = 3, End = 4 },
                new Edge() {Start = 4, End = 5 },
            };
        }
    }

    class WriterStub : IGraphWriter
    {
        public int[,] IncidenceMatrix;
        public void SaveIncidenceMatrix(string path,
                                         int[,] incidenceMatrix)
        {
            IncidenceMatrix = incidenceMatrix;
        }
    }
}

```

Пример 8.21 Пример теста с использованием заглушки и фиктивного объекта

```

using System;
using System.Collections.Generic;
using System.IO;

using NUnit.Framework;

namespace STLab08Example.Variant1
{
    [TestFixture]
    class GraphTests
    {
        [Test]
        public void GraphBFSTest()
        {
            IGraphReader reader = new ReaderStub();
            IGraphWriter writer = new WriterStub();

            Graph g = new Graph(reader, writer);
            g.LoadFromFile("");
        }
    }
}

```

```

        var result = g.BFS();

        Assert.AreEqual(5, result.Length);
    }
}
}

```

Теперь рассмотрим проблемы с тестированием классов TxtGraphFile, HtmlGraphWriter и YamlGraphFile. Самым простым решением для методов этих классов будет реализация прямого обращения к конкретному файлу на диске (например, при помощи StreamReader). Это может привести к тому, что прохождение тестов может зависеть от прав доступа к файлам или их наличия по указанному пути, а не от правильности разбора файла определенного формата.

При модульном тестировании кода необходимо стараться избегать такого рода зависимостей. Поэтому одним из решений может стать замена передачи имени файла в методы класса на использование в качестве параметра абстрактных классов TextReader и TextWriter. Данный класс является предком как для StreamReader/StreamWriter (класс для чтения/записи в файл), так и для StringReader/StringWriter (классы для чтения/записи данных в строку, как в поток), что позволяет использовать их для изоляции при модульном тестировании.

Модифицированные классы и интерфейсы представлены ниже.

Пример 8.22 Модифицированный класс Graph

```

using System;
using System.Collections.Generic;
using System.IO;

namespace STLab08Example.Variant2
{
    class Graph
    {
        // Setter Injection
        public IGraphReader Reader { get; set; }

        // Setter Injection
        public IGraphWriter Writer { get; set; }

        int[,] AdjacencyMatrix;

        // Constructor Injection
        public Graph(IGraphReader reader, IGraphWriter writer)
        {
            this.Reader = reader;
            this.Writer = writer;
            AdjacencyMatrix = null;
        }
    }
}

```

```

    }

    public void LoadFromFile(TextReader textReader)
    {
        List<Edge> edges = Reader.LoadEdgeList(textReader);

        //Convert to AdjacencyMatrix

        AdjacencyMatrix = new int[0, 0];
    }

    public void SaveToFile(TextWriter textWriter)
    {
        //Convert form AdjacencyMatrix to IncidenceMatrix

        int[,] incidenceMatrix = new int[0, 0];

        Writer.SaveIncidenceMatrix(textWriter, incidenceMatrix);
    }

    public int[] BFS()
    {
        int[] traversePath = new int[0];
        //

        return traversePath;
    }

}
}

```

Пример 8.23 Модифицированные интерфейсы IGraphReader и IGraphWriter

```

using System;
using System.Collections.Generic;
using System.IO;

namespace STLab08Example.Variant2
{
    interface IGraphReader
    {
        List<Edge> LoadEdgeList(TextReader reader);
    }

    interface IGraphWriter
    {
        void SaveIncidenceMatrix(TextWriter writer,
                                int[,] incidenceMatrix);
    }
}

```

Пример 8.24 Модифицированные прототипы реализации IGraphReader и IGraphWriter в виде классов TxtGraphFile, HtmlGraphWriter и YamlGraphFile

```
using System;
using System.Collections.Generic;
using System.IO;

namespace STLab08Example.Variant2
{
    class TxtGraphFile : IGraphReader, IGraphWriter
    {
        public List<Edge> LoadEdgeList(TextReader reader)
        {
            throw new NotImplementedException();
        }

        public void SaveIncidenceMatrix(TextWriter writer,
                                         int[,] incidenceMatrix)
        {
            throw new NotImplementedException();
        }
    }

    class YamlGraphFile : IGraphReader, IGraphWriter
    {
        public List<Edge> LoadEdgeList(string path)
        {
            throw new NotImplementedException();
        }

        public void SaveIncidenceMatrix(TextWriter writer,
                                         int[,] incidenceMatrix)
        {
            throw new NotImplementedException();
        }
    }

    class HtmlGraphWriter : IGraphWriter
    {
        public void SaveIncidenceMatrix(TextWriter writer,
                                         int[,] incidenceMatrix)
        {
            throw new NotImplementedException();
        }
    }
}
```

Пример 8.25 Модифицированный пример использования класса Graph

```
using System;
using System.Collections.Generic;
using System.IO;

namespace STLab08Example.Variant2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Set default reader and writer
            IGraphReader r = new TxtGraphFile();
            IGraphWriter w = new YamlGraphFile();

            // Create Grpah object
            Graph graph = new Graph(r, w);

            // Load graph from TXT file
            string pathTxt = "EdgeListGraph.txt";
            TextReader reader = File.OpenText(pathTxt);
            graph.LoadFromFile(reader);
            reader.Close();

            // Start BFS
            int[] traversePath = graph.BFS();

            // Save to YAML
            string pathYaml = "IncidenceMatrixGraph.yaml";
            TextWriter writer = File.CreateText(pathYaml);
            graph.SaveToFile(writer);
            writer.Close();

            // Change output formater to HTML
            graph.Writer = new HtmlGraphWriter();

            // Save to HTML
            string pathHtml = "IncidenceMatrixGraph.html";
            writer = File.CreateText(pathHtml);
            graph.SaveToFile(writer);
            writer.Close();
        }
    }
}
```

Пример 8.26 Модифицированный пример заглушки, реализующий интерфейс IGraphReader

```
using System;
using System.Collections.Generic;
```

```

using System.IO;

namespace STLab08Example.Variant2
{
    class ReaderStub : IGraphReader
    {
        public List<Edge> LoadEdgeList(TextReader reader)
        {
            return new List<Edge>()
            {
                new Edge() {Start = 1, End = 2 },
                new Edge() {Start = 2, End = 3 },
                new Edge() {Start = 2, End = 5 },
                new Edge() {Start = 3, End = 4 },
                new Edge() {Start = 4, End = 5 },
            };
        }
    }
}

```

Пример 8.27 Пример модульного теста с использованием StreamReader

```

using System;
using System.Collections.Generic;
using System.IO;

using NUnit.Framework;

namespace STLab08Example.Variant2
{
    [TestFixture]
    class TxtGraphFileTests
    {
        [Test]
        public void TxtGraphFileParseTest()
        {
            TxtGraphFile file = new TxtGraphFile();
            StreamReader reader = new StreamReader("2\n1 1\n1 2");

            var result = file.LoadEdgeList(reader);

            Assert.AreEqual(2, result.Count);
        }

        [Test]
        public void GraphSaveTest()
        {
            IGraphReader reader = new ReaderStub();
            WriterStub writer = new WriterStub();

            Graph g = new Graph(reader, writer);

```

```

        g.LoadFromFile("");
        g.SaveToFile("");

        Assert.AreEqual(5, writer.IncidenceMatrix.GetLength(0));
    }
}
}

```

3 Контрольные вопросы

- 1) Что такое тестовый двойник (test double)?
- 2) Для чего применяются тестовые двойники?
- 3) Назовите типы тестовых двойников?
- 4) Что такое объект-заглушки (Dummy Object)?
- 5) Для чего применяются объекты-заглушки (Dummy Object)?
- 6) Что такое тестовая заглушка (Test Stub)?
- 7) Для чего применяются заглушки (stub)?
- 8) Что такое тестовый агент (spy)?
- 9) Для чего применяются тестовые агенты (spy)?
- 10) Что такое поддельный объект (Fake Object)?
- 11) Для чего применяются fake-объекты?
- 12) Что такое фиктивный объект (Mock Object)?
- 13) Для чего применяются mock-объекты?
- 14) Назовите достоинства и недостатки различных тестовых двойников.
- 15) Что такое инверсия управления?
- 16) Какова цель использования инверсия управления?
- 17) Что такое внедрение зависимостей?
- 18) Для чего используется внедрение зависимостей?

4 Задание

- 1) Создать класс, реализующий граф/дерево, хранящий информацию о структуре графа на основе некоторого внутреннего представления указанного в задании из пункта 5.1.
- 2) В созданном классе реализовать один из методов обработки графа/дерева в соответствии с вариантом задания из пункта 5.2.
- 3) В качестве источника данных использовать класс, реализующий чтение из файла произвольного формата и возвращающего данные в виде, указанном в варианте задания из пункта 5.1.
- 4) Реализовать возможность сохранения данных в файл произвольного формата, с помощью отдельного класса, в формате, определяемом в соответствии с вариантом задания в пункте 5.1.

- 5) Для обработки файла создать отдельный класс/классы, реализующий методы загрузки из файла (возвращающий считанные данные в определенном формате) и сохранения в файл (сохраняющий матрицу в определенном формате, который может отличаться от внутреннего представления графа).
- 6) Протестировать класс для работы с заданным форматом файлов (при тестировании класса использовать подмену класса на `StringReader`/`StringWriter` для доступа к строковому потоку с последующей проверкой полученной выходной строки).
- 7) Протестировать класс, реализующий непосредственные вычисления (при получении данных и сохранения использовать заглушки и фиктивные объекты).
- 8) Оформить отчёт.

5 Варианты заданий

5.1 Варианты входных/выходных форматов

Варианты входных/выходных форматов данных, получаемых от `IGraphReader`/`IGraphWriter` при чтении из файлов, а также формат внутреннего представления графа указаны в таблице 8.3.

Структура хранимых данных в файлах выбирается самостоятельно.

5.2 Варианты реализуемых алгоритмов

- 1) Обход графа в ширину
- 2) Обход графа в глубину
- 3) Нахождение минимального пути в графе
- 4) Проверка на наличие циклов в графе
- 5) Проверка графа на связность
- 6) Поиск Эйлерова цикла в графе
- 7) Прямой обход дерева
- 8) Обратный обход дерева
- 9) Симметричный обход дерева

Таблица 8.3 – Варианты заданий

№	Входной формат				Внутреннее представление			Выходной формат			
	MC	МИ	СС	СР	MC	МИ	СС	MC	МИ	СС	СР
1		*			*					*	
2			*		*						*
3				*	*			*			
4	*				*				*		
5	*					*				*	
6		*				*					*
7			*			*		*			
8				*		*			*		
9	*						*				*
10		*					*	*			
11			*				*		*		
12				*			*			*	
13			*			*		*			
14				*		*			*		
15	*					*				*	
16		*				*					*
17			*				*		*		
18				*			*	*			
19	*						*				*
20		*					*			*	
21			*	*				*			
22			*		*				*		
23		*			*					*	
24	*				*						*
25			*			*					*
26		*				*				*	
27				*		*			*		
28			*			*		*			
29				*			*	*			
30			*				*		*		
31		*					*			*	
32	*						*				*

- 1) MC – Матрица смежности
- 2) МИ – Матрица инцидентности
- 3) СС – Список связности
- 4) СР – Список ребер

Лабораторная работа № 9. Автоматизация создания тестовых двойников (Test Doubles)

1 Цель работы

Изучить подход к автоматизации создания тестовых двойников (test doubles) при помощи библиотеки NSubstitute.

2 Краткая теория

Существует категория классов, которые тестировать весьма просто. Если класс зависит только от примитивных типов данных и не имеет никаких связей с другими сущностями, то достаточно создать экземпляр этого класса, проверить его путем вызовов методов и проверки ожидаемого состояния. Но количество классов, которые соответствуют данным требованиям ограничено.

При проведении автоматизированного тестирования кода, часто приходится сталкиваться с проблемой необходимости тестирования кода, часть которого еще не дописана, но какая-то функциональность есть и ее нужно протестировать. В этом случае нужно создавать заглушки для классов, интерфейсов, методов, которые используются в тестируемом методе. Делать это вручную очень утомительно, т.к. нужно писать много повторяющегося кода. Для автоматизации этого процесса существуют множество наборов библиотек (фреймворков), называемых isolation framework или mocking framework, в том числе и для языка C#. Одним из них является NSubstitute.

Создаваемые вручную макеты имеют не только преимущества, но и недостатки, один из которых заключается в растущем потоке кода, вызываемом предоставлением основанной на делегатах реализации для каждого модульного теста – особенно если при этом еще и записывать вызовы, на манер тестового шпиона. Крупнейшим же недостатком, вероятно, является то, что работа по написанию делегатов как реализаций быстро становится утомительной и подверженной ошибкам.

2.1 Характеристики NSubstitute

NSubstitute – это framework для динамической генерации mock объектов, используемых в юнит тестировании. Данный фреймворк умеет создавать mock-объекты на основе интерфейсов, делегатов и классов. Предпочтительным вариантом является использование интерфейсов и делегатов, а классы необходимо использовать с осторожностью и только в крайних случаях. При этом возможна реализация сразу нескольких интерфейсов в mock-объекте.

Если необходимо, что бы код, который мы тестируем, выполнился, но функция или метод вернул нужное значение, то имеется возможность вызвать указанную нами функцию, но для возврата значения использовать другую функцию. Т.е. имеется возможность динамически подменять вызываемые тестируемым кодом функции.

2.1 Установка NSubstitute

Для добавления возможности работы с NSubstitute необходимо установить nuget-пакет библиотеки (рисунок 9.1).

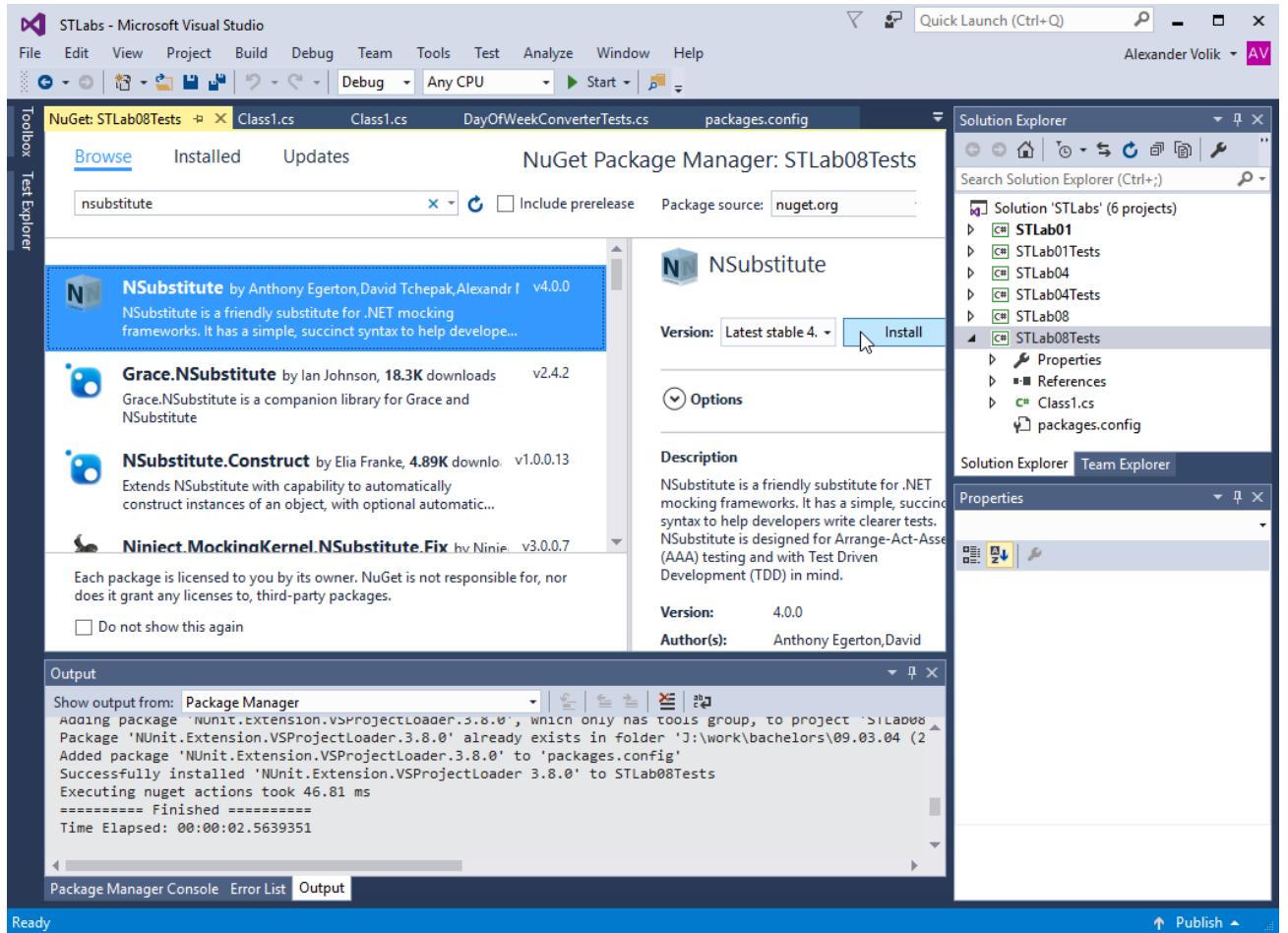


Рисунок 9.1 – Добавление библиотеки NSubstitute

Кроме этого необходимо добавить соответствующее пространство имен в список используемых при помощи директивы using.

```
using NSubstitute;
```

Помимо этого можно установить пакет NSubstitute.Analyzers.CSharp улучшающий работу библиотеки, однако для большинства небольших проектов этот пакет не нужен.

2.3 Создание и конфигурирование mock-объектов

Для создания нового mock-объекта используется статический фабричный метод For класса Substitute. Данный метод возвращает mock-объект, который может играть роль любого из тестовых двойников.

```
IInterface interfaceMock = Substitute.For<IInterface>();
```

Далее необходимо настроить работу тестовых методов mock-объекта.

Пусть имеется некоторый интерфейс ICalculator:

```
public interface ICalculator
{
    int Add(int a, int b);
    string Mode { get; set; }
}
```

Создадим для него mock-объект и настроим его работу.

Для создания mock-объекта достаточно просто вызывать метод For:

```
ICalculator calculator = Substitute.For<ICalculator>();
```

Теперь можно определить возвращаемое значение для вызова метода. Для этого достаточно указать параметры для вызова в оригинальном методе интерфейса и в методе Returns (доступном у оригинального метода mock-объекта) указать возвращаемое значение:

```
calculator.Add(1, 2).Returns(3);
```

Теперь при вызове метода mock-объекта с указанными параметрами он вернёт соответствующее значение:

```
Assert.That(calculator.Add(1, 2), Is.EqualTo(3));
```

Так же мы можем проверить, что у mock-объекта были вызваны определённые методы и не было вызовов остальных. Для этого используется методы Received/ DidNotReceive mock-объекта:

```
calculator.Add(1, 2);
calculator.Received().Add(1, 2);
calculator.DidNotReceive().Add(5, 7);
```

Если проверка в Received() не проходит, то NSubstitute указывает, почему это произошло. Например, при вызове метода с параметрами Add(4, 7) и Add(1, 5) будет показано подобное сообщение:

```
NSubstitute.Exceptions.ReceivedCallsException : Expected to receive a
call matching:
    Add(1, 2)
Actually received no matching calls.
Received 2 non-matching calls (non-matching arguments indicated with '*' 
characters):
    Add(*4*, *7*)
    Add(1, *5*)
```

Помимо этого можно указывать возвращаемые значения не только для методов, но и для свойств:

```
calculator.Mode.Returns("DEC");
Assert.That(calculator.Mode, Is.EqualTo("DEC"));

calculator.Mode = "HEX";
Assert.That(calculator.Mode, Is.EqualTo("HEX"));
```

Так же NSubstitute поддерживает сопоставление (matching) аргументов по образцу при проверке вызовов. В тех случаях, когда при проверке неважно значение кого-либо аргумента, либо необходимо проверить определённый критерий для аргумента можно воспользоваться статическими методами класс Arg. Например, когда передаваемое значение не важно можно использовать метод Any, а для проверки что было передано положительное число воспользоваться методом Is с указанием лямбда-функции проверки:

```
calculator.Add(10, -5);
calculator.Received().Add(10, Arg.Any<int>());
calculator.Received().Add(10, Arg.Is<int>(x => x < 0));
```

Также, помимо сопоставления, возможна передача аргументов в метод Returns() для описания некоторого поведения вне mock-объекта. Например, можно выполнить обработку входных значений.

```
calculator
    .Add(Arg.Any<int>(), Arg.Any<int>())
    .Returns(x => (int)x[0] + (int)x[1]);
Assert.That(calculator.Add(5, 10), Is.EqualTo(15));
```

Метод Returns() может быть вызван с несколькими аргументами, которые буду последовательно возвращаться при каждом новом вызове функции:

```
calculator.Mode.Returns("HEX", "DEC", "BIN");
Assert.That(calculator.Mode, Is.EqualTo("HEX"));
Assert.That(calculator.Mode, Is.EqualTo("DEC"));
Assert.That(calculator.Mode, Is.EqualTo("BIN"));
```

Кроме возвращения определённых значений можно вызывать исключения при работе тестируемых классов:

```
calculator.Add(-1, -1).Returns(x => { throw new Exception(); });
```

Более подробно с работой с библиотекой можно ознакомиться на официальном сайте <https://nsubstitute.github.io>.

2.3 Пример создания мок-объектов

Возьмём класс DataManager из прошлой лабораторной работы для тестирования (пример 8.1).

Пример 8.1 Реализация тестируемого класса

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace STLab09a
{
    public class DataManager
    {
        private ITypedFile dataFile;
        private List<string> data;
        private int recordCount;

        public DataManager(ITypedFile dataFile)
        {
            if (dataFile == null)
            {
                throw new ArgumentNullException("dataFile");
            }
            this.dataFile = dataFile;
            this.recordCount = 0;
            this.data = new List<string>();
        }

        public void AddData(string record)
```

```

    {
        if (record == null)
            throw new ArgumentNullException("record");
        this.recordCount++;
        this.data.Add(record);
    }

    public int RecordCount
    {
        get { return this.recordCount; }
    }

    public string GetFileRecord(int id)
    {
        if (id < 0)
            throw new ArgumentException("id");

        return dataFile.GetRecord(id);
    }

    public void Save()
    {
        this.dataFile.Save();
    }

    internal ITypedFile DataFile
    {
        get { return this.dataFile; }
    }
}

}

```

Рассмотрим тестирование метода и получающего запись из файла при помощи `GetFileRecord`. Тестовый метод, аналогичный примеру из предыдущей работы представлен в примере 9.2.

Пример 9.2 Тестирование `GetFileRecord` с помощью NSubstitute

```

public void GetFileRecordTest()
{
    // Arrange: create mocks, test data and expected methods calls
    ITypedFile dataFile = Substitute.For<ITypedFile>();

    int id = 0;
    string expected = "0";

    dataFile
        .GetRecord(id)
        .Returns("0");
}

```

```

    DataManager dataManager = new DataManager(dataFile);

    // Act: invoke testing method
    string actual = dataManager.GetFileRecord(id);

    // Assert: verify expectations
    Assert.AreEqual(expected, actual);
}

```

Для проверки вызова метода Save тест будет выглядеть, как показано в примере 9.3.

Пример 9.3 Реализация тестирования Save

```

[Test()]
public void SaveTest()
{
    // Arrange: create mocks,test data and expected methods calls
    ITypedFile dataFile = Substitute.For<ITypedFile>();

    DataManager dataManager = new DataManager(dataFile);

    // Act: invoke testing method
    dataManager.Save();

    // Assert: Implicit assertion
    dataFile.Received().Save();
}

```

Если требуется возвращать различные значения, в зависимости от значения входного параметра, то для этого можно воспользоваться несколькими описаниями вызовов с различными вариантами аргументов и возвращаемых значений (примере 9.4).

Пример 9.4 Тестирование GetFileRecord с несколькими значениями

```

[Test()]
public void GetFileRecordMultipleValuesTest()
{
    // Arrange: create mocks,test data and expected methods calls
    ITypedFile dataFile = Substitute.For<ITypedFile>();

    dataFile.GetRecord(0).Returns("0");
    dataFile.GetRecord(1).Returns("1");
    dataFile.GetRecord(2).Returns("2");
    dataFile.GetRecord(4).Returns("4");

    DataManager dataManager = new DataManager(dataFile);

    // Act: invoke testing method
}

```

```

    string actual = dataManager.GetFileRecord(0);

    // Assert: verify expectations
    Assert.AreEqual("0", actual);

    // Act: invoke testing method
    actual = dataManager.GetFileRecord(1);
    // Assert: verify expectations
    Assert.AreEqual("1", actual);
}

```

Кроме возвращения определённых значений можно вызывать исключения при работе тестируемых классов (пример 9.5).

Пример 9.5 Тест GetFileRecord с вызовом исключений

```

[Test()]
public void GetFileRecordTest_ThrowsException()
{
    // Arrange: create mocks,test data and expected methods calls
    ITypedFile dataFile = Substitute.For<ITypedFile>();

    int id = -1;

    dataFile
        .GetRecord(id)
        .Returns((x) => { throw new ArgumentException("ID"); });

    DataManager dataManager = new DataManager(dataFile);

    // Act + Assert: invoke testing method and verify expectations
    Assert.Throws< ArgumentException >(
        () => dataManager.GetFileRecord(id));
}

```

3 Контрольные вопросы

- 1) Что такое Isolation Framework?
- 2) Какие виды mock-объекты поддерживает NSubstitute?
- 3) В чем отличие основных видов?
- 4) Как создать mock-объект?
- 5) Как настроить возвращаемые значения для метода mock-объекта?
- 6) Как указать множество возвращаемых значений для метода mock-объекта?
- 7) Как проверить, что методы был вызван с определёнными параметрами?
- 8) Как проверить, что метод не был вызван?
- 9) Как указать, что метод должен вернуть исключение?

10) Как настроить возвращаемые значения для свойства mock-объекта?

4 Задание

- 1) На основе класса из предыдущей лабораторной работы (вариант задания оставить прежним) создать тестирующий проект, создающий тестовых двойников при помощи NSubstitute
- 2) Протестировать класс.
- 3) Оформить отчёт.

Лабораторная работа № 10. Упрощение создания тестов при помощи библиотеки Fluent Assertions

1 Цель работы

Изучить подход к созданию тестов с помощью библиотеки Fluent Assertions.

2 Краткая теория

Одной из причин неудобства создания и поддержки тестов является отсутствие внятных сообщений о причине падения тестов. Чаще всего вам приходится использовать точки остановки и отладчик для понимания, что произошло с тестом, и почему он перестал проходить.

Хорошо, если вы можете написать тест с одной проверкой, тогда проблема с определением причины будет автоматически решена. Но если в тесте присутствует несколько проверок (например, проверяются различные поля объекта содержащего результат), то тестовый фреймворк, получив первую же ошибку, не станет проверять остальные. И в этом случае вам придется дополнительно выяснять, какая именно проверка привела к падению теста.

Библиотека Fluent Assertions помогает бороться с этой проблемой. Она не только предоставляет понятные имена для методов проверки (assertion), но и заботится о том, что бы сообщение об ошибке было как можно более информативным.

2.1 Характеристики Fluent Assertions

Библиотека Fluent Assertions обладает мощными средствами для проведения проверок с удобными и подробными сообщениями об ошибках.

Например, тест:

```
string accountNumber = "123";
accountNumber.Should().Be("120");
```

выведет на экран:

```
Expected accountNumber to be "132", but "123" differs near "23" (index 1).
```

А если длина строки результата больше восьми символов, то она начинается с новой строки:

```
string accountNumber = "1234567890";
accountNumber.Should().Be("0987654321");
```

выведет на экран:

```
Expected accountNumber to be  
"0987654321", but  
"1234567890" differs near "123" (index 0).
```

Если же стандартного вывода не достаточно, то каждый из методов позволяет задать произвольное описание, поддерживающее форматирование, аналогичное методам String.Format.

Например, тест:

```
var numbers = new[] { 1, 2, 3 };  
numbers.Should().Contain(item => item > 3, "at least {0} item should be  
larger than 3", 1);
```

упадет с сообщением:

```
Expected numbers {1, 2, 3} to have an item matching (item > 3) because  
at least 1 item should be larger than 3.
```

Библиотека Fluent Assertions не является самостоятельным тестовым фреймворком, а работает совместно с существующими, обеспечивая удобство и простоту написания тестов. Она представляет собой набор методов расширений, которые позволяют в более естественном виде описывать ожидаемые результаты. В основе работы библиотеки лежит паттерн Fluent Interface и большое количество методов расширений, позволяющих организовывать хорошо читаемые конструкции проверок.

Поддерживает следующий список тестовых фреймворков:

- MSTest (Visual Studio 2010, 2012 Update 2, 2013 and 2015)
- MSTest V2 (Visual Studio 2017, Visual Studio 2019)
- NUnit
- XUnit
- XUnit2
- MBUnit
- Gallio
- NSpec
- MSpec

2.2 Установка Fluent Assertions

Для добавления возможности работы с Fluent Assertions необходимо установить nuget-пакет библиотеки (рисунок 10.1).

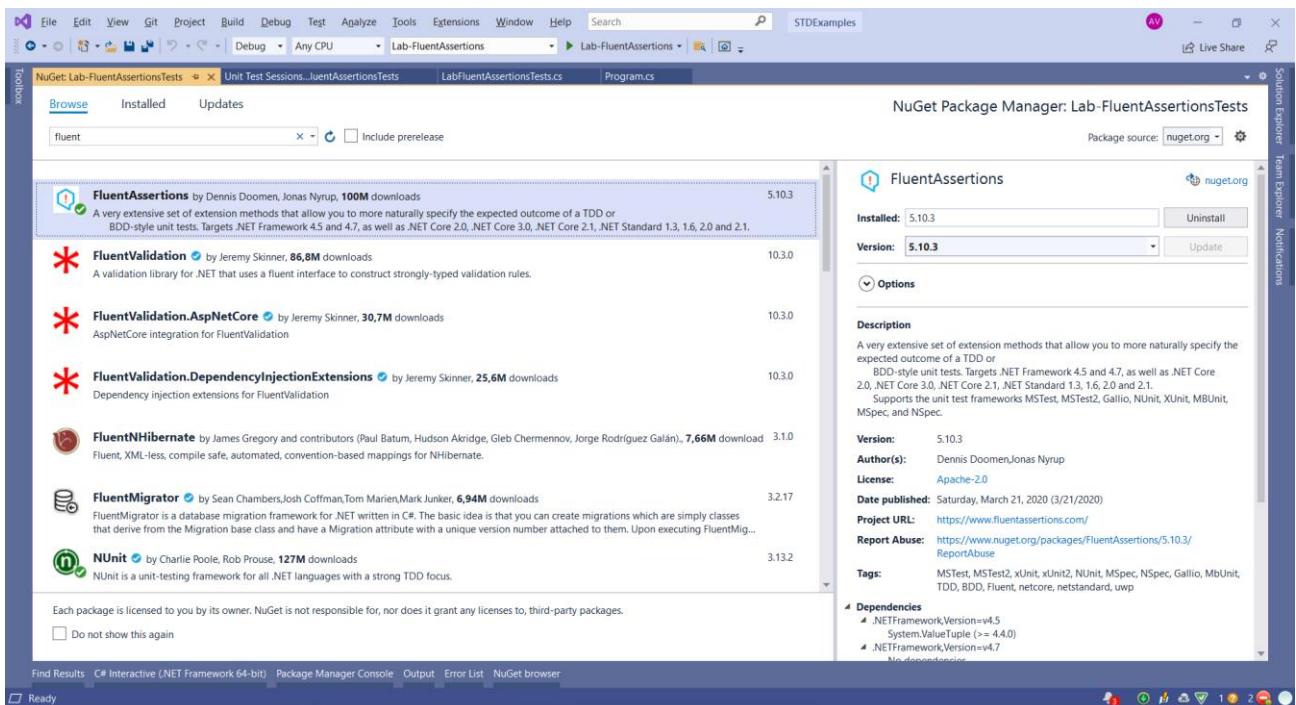


Рисунок 10.1 – Добавление библиотеки Fluent Assertions

Кроме этого необходимо добавить соответствующее пространство имен в список используемых при помощи директивы `using`.

```
using FluentAssertions;
```

Помимо этого для написания и запуска тестов также необходимо установить пакет `NUnit`, обеспечивающий работу библиотеки.

2.3 Создание тестов с помощью библиотеки Fluent Assertions

2.3.1 Общие принципы работы

Библиотека `Fluent Assertions` представляет собой набор методов расширений, которые позволяют в более естественном виде описывать ожидаемые результаты. Это соответствует стилям написания тестов в подходах TDD или BDD.

Более подробно с принципами работы с библиотекой можно ознакомиться на официальном сайте <https://fluentassertions.com>.

Подключение пространства имен `FluentAssertions` добавляет большое количество методов расширения в текущую область видимости (scope).

Например, что бы проверить, что строка начинается с некоторого префикса, заканчивается на некоторый суффикс и содержит некоторую подстроку, а также содержит определенное количество символов, можно использовать следующую конструкцию:

```
string actual = "ABCDEFGHI";
actual.Should().StartWith("AB").And.EndWith("HI").And.Contain("EF").And.
HaveLength(9);
```

Часто бывает удобно организовывать цепочки проверок, помещая логические блоки на отдельные строки. Предыдущий пример можно переписать в следующем виде:

```
string actual = "ABCDEFGHI";
actual.Should()
    .StartWith("AB").And
    .EndWith("HI").And
    .Contain("EF").And
    .HaveLength(9);
```

А для проверки длины коллекции и соответствие элементов некоторому условию, можно построит следующее выражение:

```
IEnumerable<int> numbers = new[] { 1, 2, 3 };

numbers.Should().OnlyContain(n => n > 0);
numbers.Should().HaveCount(4, "because we thought we put four items in
the collection");
```

Для данного примера сообщение об ошибке будет выглядеть следующим образом:

The nice thing about the second failing example is that it will throw an exception with the message

```
Expected numbers to contain 4 item(s) because we thought we put four
items in the collection, but found 3.
```

Для проверки возникновения исключения можно использовать следующий код:

```
[Test]
public void SampleTest07()
{
    var data = new Dictionary<int, int>()
    {
        {1, 1},
        {2, 15},
        {3, 18}
    };

    int Calc(IDictionary<int, int> data)
    {
        return data[0] * 3;
    }
}
```

```

Action action = () => Calc(data);

action
    .Should().Throw<KeyNotFoundException>()
    .WithMessage("*was not present in the dictionary*");
}

```

Помимо этого удобным способом можно проверять целые цепочки в коллекциях и графах объектов:

```

class Data
{
    public int ValueProerty { get; set; }
    public string Message { get; set; }
}

[TestMethod]
public void SampleTest08()
{
    var data1 = new Data()
    {
        ValueProerty = 2,
        Message = "Message from data1"
    };

    var dictionary = new Dictionary<int, Data>()
    {
        {1, data1},
        {
            2, new Data()
            {
                ValueProerty = 3
            }
        },
    };

    dictionary.Should()
        .ContainValue(data1)
        .Which.ValueProerty.Should().BeGreaterThan(0);

    data1.Should()
        .BeOfType<Data>()
        .Which.Message.Should().Be("Message from data1");
}

```

Такой подход позволяет создавать более читаемые модульные тесты.

2.3.2 Базовые проверки (assertions)

Приведенные ниже проверки доступны для всех типов объектов:

```

object theObject = null;
theObject.Should().BeNull("because the value is null");
theObject.Should().NotBeNull();

theObject = "whatever";
theObject.Should().BeOfType<string>("because a {0} is set",
typeof(string));
theObject.Should().BeOfType(typeof(string), "because a {0} is set",
typeof(string));

```

Иногда полезным бывает сначала проверить тип объекта, а затем продолжить с дополнительными проверками с учетом приведения к конкретному типу. Для обращения к свойствам используется свойство-расширение Which, которое позволяет удобно выставлять цепочки.

```

var exception = new Exception("Message");

exception.Should().BeOfType<Exception>()
    .Which.Message.Should().Be("Message");

```

Ниже приведено еще несколько примеров проверки типов:

```

var ex = new ArgumentException();
ex.Should().BeAssignableTo<Exception>("because it is an exception");
ex.Should().NotBeAssignableTo<DateTime>("because it is an exception");

var dummy = new Object();
dummy.Should().Match(d => (d.ToString() == "System.Object"));
dummy.Should().Match<string>(d => (d == "System.Object"));
dummy.Should().Match((string d) => (d == "System.Object"));

```

2.3.3 Проверки строковых значений

Для выполнения различных проверок строк имеется большой набор методов. Ниже приведены примеры использования, назначение которых понятно из названий.

```

string theString = "";
theString.Should().NotBeNull();
theString.Should().BeNull();
theString.Should().BeEmpty();
theString.Should().NotBeEmpty("because the string is not empty");
theString.Should().HaveLength(0);
theString.Should().BeNullOrEmpty(); // either null, empty or
// whitespace only
theString.Should().NotBeNullOrEmpty();

theString = "This is a String";
theString.Should().Be("This is a String");
theString.Should().NotBe("This is another String");

```

```

theString.Should().BeEquivalentTo("THIS IS A STRING");
theString.Should().NotBeEquivalentTo("THIS IS ANOTHER STRING");

theString.Should().BeOneOf(
    "This is a String",
    "THIS IS A STRING"
);

theString.Should().Contain("is a");
theString.Should().Contain("is a", Exactly.Once());
theString.Should().Contain("is a", AtLeast.Twice());
theString.Should().Contain("is a", MoreThan.Thrice());
theString.Should().Contain("is a", AtMost.Times(5));
theString.Should().Contain("is a", LessThan.Twice());
theString.Should().ContainAll("should", "contain", "all", "of",
    "these");
theString.Should().ContainAny("any", "of", "these", "will", "do");
theString.Should().NotContain("is a");
theString.Should().NotContainAll("can", "contain", "some", "but", "not",
    "all");
theString.Should().NotContainAny("can't", "contain", "any", "of",
    "these");
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING");
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING",
    Exactly.Once());
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING",
    AtLeast.Twice());
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING",
    MoreThan.Thrice());
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING",
    AtMost.Times(5));
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING",
    LessThan.Twice());
theString.Should().NotContainEquivalentOf("HeRe ThE CaSiNg Is IgNoReD As
WeLl");

theString.Should().StartWith("This");
theString.Should().NotStartWith("This");
theString.Should().StartWithEquivalent("this");
theString.Should().NotStartWithEquivalentOf("this");

theString.Should().EndWith("a String");
theString.Should().NotEndWith("a String");
theString.Should().EndWithEquivalent("a string");
theString.Should().NotEndWithEquivalentOf("a string");

```

Помимо этого поддерживаются знаки подстановки (wildcards) как для регистрозависимого, так и для регистронезависимого случаев:

```

var emailAddress = "info@example.com";
var homeAddress = "135, Krasnaya st., Krasnodar";

```

```

emailAddress.Should().Match("*@*.com");
homeAddress.Should().NotMatch("*@*.com");

emailAddress.Should().MatchEquivalentOf("*@*.COM");
homeAddress.Should().NotMatchEquivalentOf("*@*.COM");

```

Так поддерживаются регулярные выражения:

```

var homeAddress = "135, Krasnaya st., Krasnodar";

homeAddress.Should().MatchRegex("\\d+.*\\sst\\..+$");
homeAddress.Should().NotMatchRegex(".*earth.*");

```

2.3.4 Проверки коллекций

Коллекции в .NET очень разнообразны и обладают огромным набором методов. Поэтому библиотека Fluent Assertions имеет соответствующее количество проверок. Ниже приведены примеры проверок, назначения которых понятны из названий:

```

IEnumerable<int> collection = new[] { 1, 2, 5, 8 };
IEnumerable<int> oneItemCollection = new[] { 42 };
IEnumerable<int> emptyCollection = new int[0];

collection.Should().NotBeEmpty()
    .And.HaveCount(4)
    .And.ContainInOrder(new[] { 2, 5 })
    .And.ContainItemsAssignableTo<int>();

collection.Should().Equal(new List<int> { 1, 2, 5, 8 });
collection.Should().Equal(1, 2, 5, 8);
collection.Should().NotEqual(new[] { 8, 2, 3, 5 });
collection.Should().BeEquivalentTo(8, 2, 1, 5);
collection.Should().NotBeEquivalentTo(new[] { 8, 2, 3, 5 });

collection.Should().HaveCount(c => c > 3)
    .And.OnlyHaveUniqueItems();

collection.Should().HaveCountGreaterThan(3);
collection.Should().HaveCountGreaterOrEqualTo(4);
collection.Should().HaveCountLessThanOrEqualTo(4);
collection.Should().HaveCountLessThan(5);
collection.Should().NotHaveCount(3);
collection.Should().HaveSameCount(new[] { 6, 2, 0, 5 });
collection.Should().NotHaveSameCount(new[] { 6, 2, 0 });

collection.Should().StartWith(1);
collection.Should().StartWith(new[] { 1, 2 });
collection.Should().EndWith(8);

```

```

collection.Should().EndWith(new[] { 5, 8 });

collection.Should().BeSubsetOf(new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, });

oneItemCollection.Should().ContainSingle();
collection.Should().ContainSingle(x => x > 6);
collection.Should().Contain(8)
    .And.HaveElementAt(2, 5)
    .And.NotBeSubsetOf(new[] { 11, 56 });

collection.Should().Contain(x => x > 3);
collection.Should().Contain(collection, "", 5, 6); // It should contain
the original items, plus 5 and 6.

collection.Should().OnlyContain(x => x < 10);
collection.Should().ContainItemsAssignableTo<int>();

collection.Should().ContainInOrder(new[] { 1, 5, 8 });

collection.Should().NotContain(82);
collection.Should().NotContain(new[] { 82, 83 });
collection.Should().NotContainNulls();
collection.Should().NotContain(x => x > 10);

object boxedValue = 2;
collection.Should().ContainEquivalentOf(boxedValue); // Compared by
object equivalence

const int element = 5;
const int successor = 8;
const int predecessor = 2;
collection.Should().HaveElementPreceding(successor, element);
collection.Should().HaveElementSucceeding(predecessor, element);

emptyCollection.Should().BeEmpty();
emptyCollection.Should().BeNullOrEmpty();
collection.Should().NotBeNullOrEmpty();

IEnumerable<int> otherCollection = new[] { 1, 2, 5, 8, 1 };
IEnumerable<int> anotherCollection = new[] { 10, 20, 50, 80, 10 };
collection.Should().IntersectWith(otherCollection);
collection.Should().NotIntersectWith(anotherCollection);

IEnumerable<int> ascendingOrderCollection = new[] { 3, 5, 13 };
IEnumerable<int> desceninOrderCollection = new[] { 3, 2, 1 };

ascendingOrderCollection.Should().BeInAscendingOrder();
desceninOrderCollection.Should().BeInDescendingOrder();
desceninOrderCollection.Should().NotBeInAscendingOrder();
ascendingOrderCollection.Should().NotBeInDescendingOrder();

```

```
IEnumerable<string> stringCollection = new[] { "build succeeded", "test failed" };
stringCollection.Should().ContainMatch("* failed");
```

Кроме этого поддерживается работа с коллекциями составных типов. Для их работы необходимо, что бы тип реализовал интерфейс IComparable. Но кроме этого можно использовать и явное указания свойств (property) для проверок:

```
class CustomData
{
    public int Value { get; set; }
    public string Message { get; set; }
}

[TestMethod]
public void SampleTest16()
{
    IEnumerable<CustomData> collection = new[]
    {
        new CustomData()
        {
            Value = 1,
            Message = "123456789",
        },
        new CustomData()
        {
            Value = 3,
            Message = "000",
        },
    };

    collection.Should().BeInAscendingOrder(x => x.Value);
    collection.Should().BeInDescendingOrder(x => x.Message);
    collection.Should().NotBeInAscendingOrder(x => x.Message);
    collection.Should().NotBeInDescendingOrder(x => x.Value);
}
```

3 Контрольные вопросы

- 1) Что представляет собой библиотека Fluent Assertions?
- 2) Какие преимущества дает использования библиотеки Fluent Assertions по сравнению с обычными тестовыми фреймворками?
- 3) Какие методы написания тестов поддерживает библиотека?

4 Задание

- 1) На основе класса из предыдущей лабораторной работы (вариант задания оставить прежним) создать тестирующий проект, использующий библиотеку Fluent Assertions
- 2) Протестировать проект.
- 3) Оформить отчёт.

Лабораторная работа № 11. Мутационное тестирование

1 Цель работы

Изучить принципы работы мутационного тестирования и использование библиотеки Stryker.NET для создания мутационных тестов.

2 Краткая теория

2.1 Понятие мутационного тестирования

Мутационное тестирование — это метод тестирования программного обеспечения, основанный на всевозможных изменениях исходного кода и проверке реакции на эти изменения набора автоматических тестов. Если тесты после изменения кода успешно выполняются, значит либо код не покрыт тестами, либо написанные тесты неэффективны. Критерий, определяющий эффективность набора автоматических тестов, называется Mutation Score Indicator (MSI).

Мутационное тестирование выполняется путем изменения или изменения фрагмента кода, чтобы проверить убедиться, что тестовые сценарии способны обнаружить ошибки/дефект.

Юнит тесты помогают нам удостовериться, что код работает так, как мы этого хотим. Одной из метрик тестов является процент покрытия строк кода (Line Code Coverage).

Но насколько корректен данный показатель? Имеет ли он практический смысл, и можем ли мы ему доверять? Ведь если мы удалим все Assert строки из тестов, или просто заменим их на Assert.AreEqual(1, 1), то по-прежнему будем иметь 100% Code Coverage, при этом тесты ровным счетом не будут тестировать ничего.

Введем некоторые понятия из теории мутационного тестирования:

Для применения этой технологии у нас, очевидно, должен быть исходный код (source code), некоторый набор тестов (для простоты будем говорить о модульных тестах – unit tests).

После этого можно начинать изменять отдельные части исходного кода и смотреть, как реагируют на это тесты.

Одно изменение исходного кода будем называть Мутацией (Mutation). Например, изменение бинарного оператора "+" на бинарный "-" является мутацией кода.

Результатом мутации является Мутант (Mutant) — то есть это новый мутированный исходный код.

Каждая мутация любого оператора в вашем коде (а их сотни) приводит к новому мутанту, для которого должны быть запущены тесты.

Кроме изменения "+" на "-", существует множество других мутационных операторов (Mutation Operator, Mutator) — отрицание условий, изменение возвращаемого значения функции, удаление строк кода и т.д.

Библиотеки мутационного тестирования создают множество мутантов из вашего кода, для каждого из них запускают тесты и проверяют, выполнились они успешно или нет. Если тесты упали — значит всё хорошо, они отреагировали на изменение в коде и поймали ошибку. Такой мутант считается убитым (**Killed mutant**). Если тесты выполнились успешно после мутации — это говорит о том, что либо ваш код не покрыт в этом месте тестами вовсе, либо тесты, покрывающие мутированную строку, неэффективны и в недостаточной степени тестируют данный участок кода. Такой мутант называется выжившим (**Survived, Escaped Mutant**).

Важно понимать, что мутационное тестирование это не хаотичное преобразование кода, а абсолютно предсказуемый и понятный процесс, который, при наличии одинаковых входных мутационных операторов, всегда выдает один и тот же список мутаций и результирующие метрики на одинаковом тестируемом исходном коде.

2.2 Цели мутационного тестирования

Эдсгер Вибе Дейкстра: Тестирование выявляет наличие, а не отсутствие ошибок.

Даже для обнаружения имеющихся ошибок необходимо написать немало тестов, чтобы внести изменения и выпустить максимально надежный код.

О недостаточном количестве проведенных тестов вы сможете узнать двумя разными способами. Во-первых, разработанное ПО в итоге не будет соответствовать требованиям, и его пользователи не получат того, за что заплатили. Избежать подобного сценария поможет внимательное ревью кода и его сопоставление с изначальными требованиями. Помимо этого, следует предоставить пользователям ранний доступ к ПО. Во-вторых, не каждая строка и ветвь кода проверяются набором тестов. Это легко определить с помощью целого арсенала существующих инструментов покрытия кода, часто встроенных в исполнитель модульных тестов или используемую библиотеку.

Покрытие кода или тестовое покрытие — это показатель степени выполнения кода во время тестирования. Измерить его можно разными способами. В целом, 100% покрытие проекта означает, что модульные тесты проверили каждый участок кода. Достижению высокого процента, безусловно, способствует техника разработки через тестирование. Если вы

создадите тест до написания самого кода, то почти без усилий добьетесь 100%.

Можно предположить, что при высоком проценте покрытия вероятность появления ошибки резко снижается. Но если пойти дальше и убрать все утверждения из модульных тестов, то покрытие останется точно таким же. То есть теперь у вас набор тестов со 100% покрытием, но фактической пользы от него нет. Уберите все утверждения из модульных тестов, и покрытие кода останется тем же самым.

По-настоящему качественный набор тестов обеспечит семантическую стабильность ПО: если при модификации кода вы меняете его содержание, то, по крайней мере, один тест-кейс не пройдет, сигнализируя о семантическом изменении. Это как раз ожидаемое поведение всех наборов тестов, которого зачастую не так легко добиться.

Как же создать качественный набор тестов? И как убедиться в том, что вы не просто выполняете каждую инструкцию, но при этом еще и проверяете ее? Для этих целей и служит мутационное тестирование.

2.3 Типы мутаций

Типы вносимых в код изменений зависят от используемого языка и его парадигмы.

Список возможных мутаций бесконечен, так что ограничимся обзором лишь некоторых из них:

- замена арифметических операторов, например * на +, как в ранее рассмотренном фрагменте кода;
- замена логических выражений на true или false;
- замена константы другим значением: зачастую к числам добавляется 1 или -1, а к строкам — "XX";
- замена логических операторов, например and на or и наоборот;
- замена логических отношений, а именно < на <=;
- замена break на continue;
- замена условных конструкций if и while, например if (condition) на if (not condition);
- блокировка исключений;
- удаление вызовов функций и других инструкций;
- удаление вызовов base();
- замена модификаторов доступа, к примеру public на private.

2.4 Анализ результатов

Если после внесения намеренной ошибки набор тестов проваливается, значит, мутант обнаружен и уничтожен. Но если тесты

проходят, а мутант остается незамеченным — он выжил. Выживший мутант указывает на отсутствие нужного тест-кейса или на необходимость улучшения уже имеющегося.

В редких случаях мутация делает код некомпилируемым или приводит к выполнению бесконечного цикла, в связи с чем стоит рассмотреть возможные последствия замены `break` на `continue`. Мутанты, из-за которых части кода становятся бессмысленными, малоэффективны. Большинство инструментов используют тайм-ауты для обработки бесконечных циклов и обычно отмечают таких мутантов в статистике.

Иногда мутация совсем не меняет поведение кода и остается незамеченной. Речь идет о так называемых идентичных мутантах, наличие которых сильно затрудняет практическое применение мутационного тестирования. Обнаружение таких мутантов представляет собой отдельную научную область, подкрепленную соответствующей проектной документацией.

Об успешности сеанса мутационного тестирования можно судить по показателю мутации, который отражает соотношение убитых и сгенерированных мутантов. Если все мутанты обезврежены, то он равняется 100%, и вы можете быть уверены в надежности вашего набора тестов.

2.5 Недостатки мутационного тестирования

Даже ПО средней сложности может иметь несколько сотен возможных мутаций, каждая из которых повлечет за собой очередное выполнение набора тестов.

Если для выполнения модульных тестов требуется 30 секунд, то несколько сотен мутаций увеличат это время до нескольких часов, и это без учета времени компиляции в таких языках, как C++, C# или Java. Для решения этой проблемы многие инструменты задействуют различные формы многопроцессорной обработки, в результате чего длительность выполнения определяется уже производительностью системы. Так или иначе мутационное тестирование является дорогостоящей процедурой.

2.2 Библиотека Stryker.NET

Рассмотрим пример использования библиотеки Stryker.NET. Создадим решение *MutationTesting* с проектом библиотеки *DemoMath* and и тестовый проект для нее *DemoMath.Tests*. Проекты должны быть созданы для среды выполнения .Net Core (.Net 5). Код проектов можно взять из приложения Б.

2.2.1 Установка Stryker.NET

Для начала необходимо создать манифест инструментов (tool manifest) для решения *MutationTesting* запустив команду в корне решения (solution) (см. рисунок 12.1).

```
dotnet new tool-manifest
```

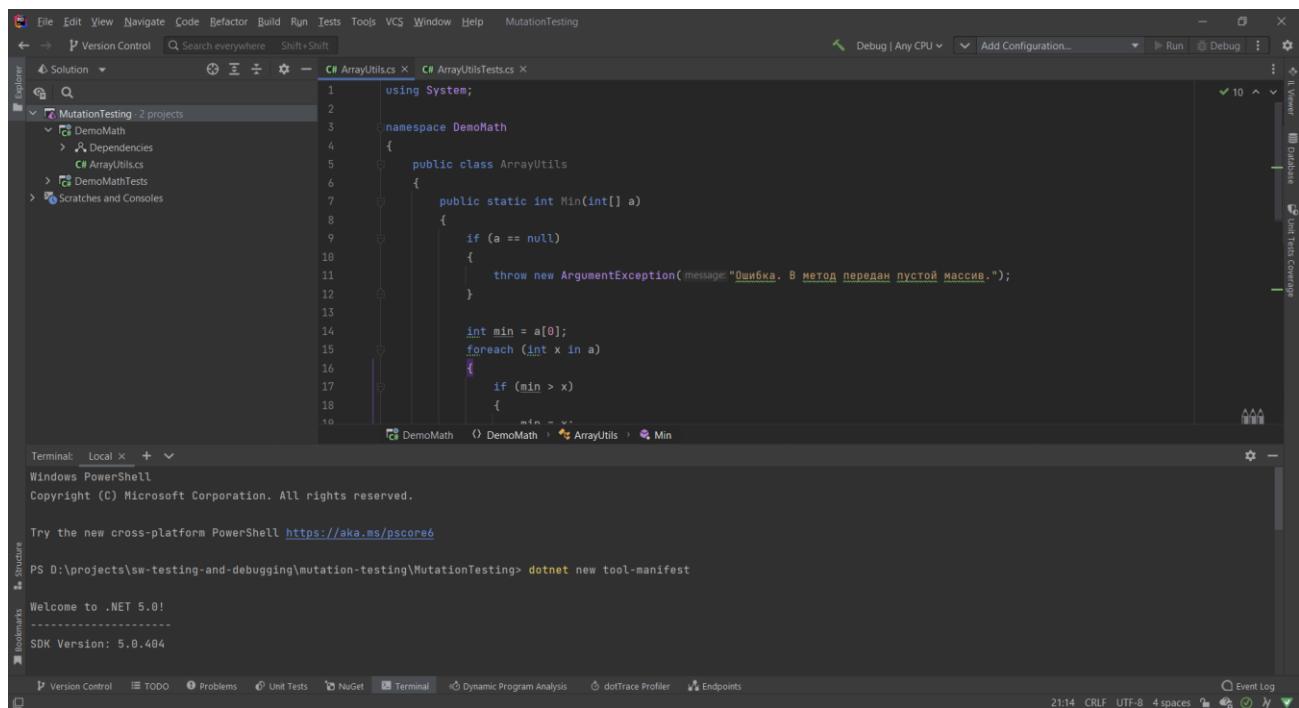


Рисунок 12.1 – Создание манифеста

После выполнения команды в каталоге решения появится папка *.config* содержащая файл *dotnet-tools.json*.

После этого подготовительного шага можно приступать к непосредственной установки Stryker.NET. Для этого необходимо выполнить команду (рисунок 12.2).

```
dotnet tool install dotnet-stryker
```

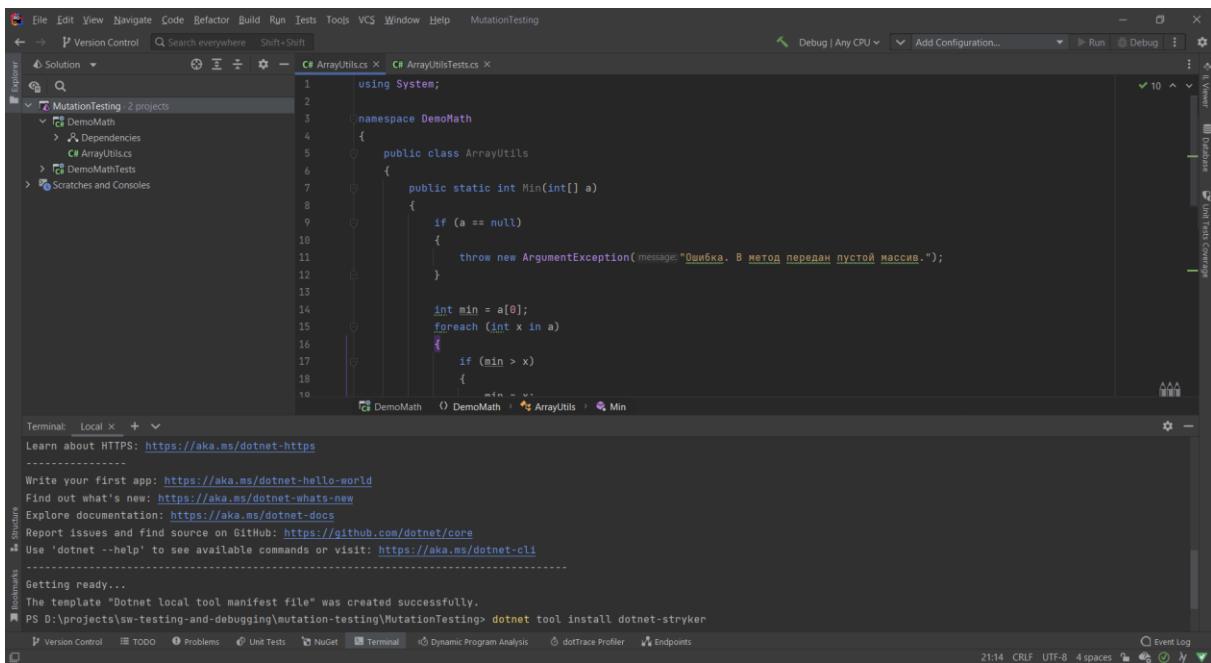


Рисунок 12.2 – Установка Stryker.NET

После успешной установки будет показано соответствующее сообщение (рисунок 12.3) и в файле манифеста появится новая запись.

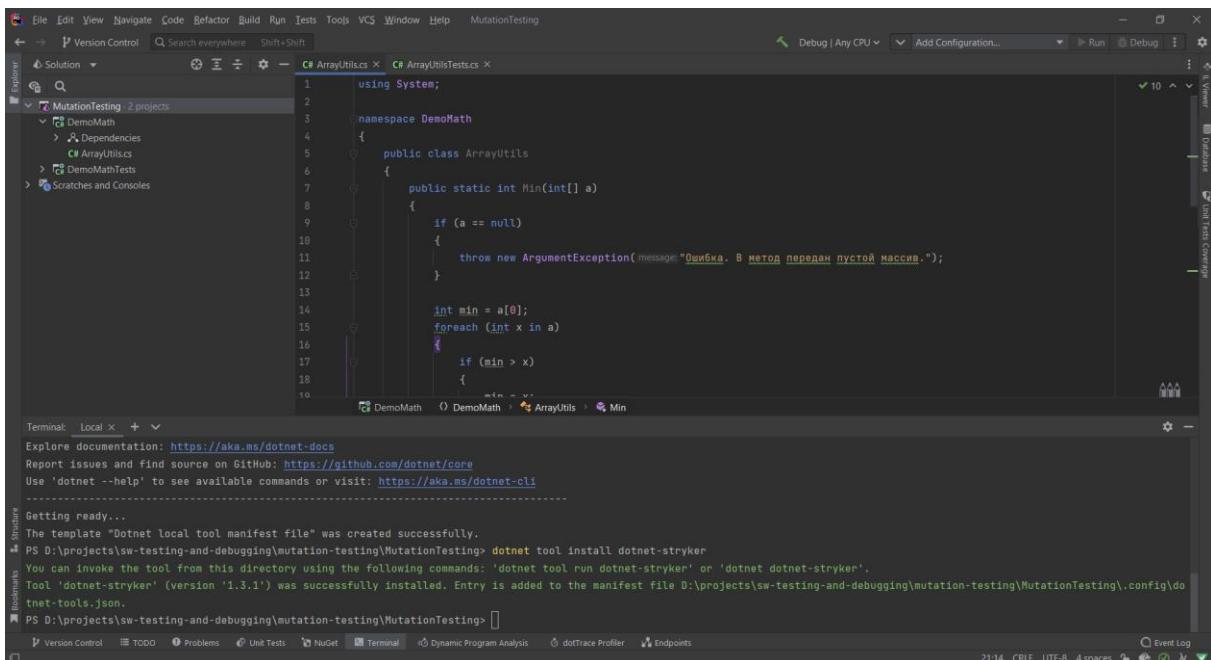


Рисунок 12.3 – Успешная установка Stryker.NET

Теперь можно запустить мутационное тестирование проекта. Для этого необходимо запустить команду внутри тестового проекта (рисунок 12.4).

dotnet stryker

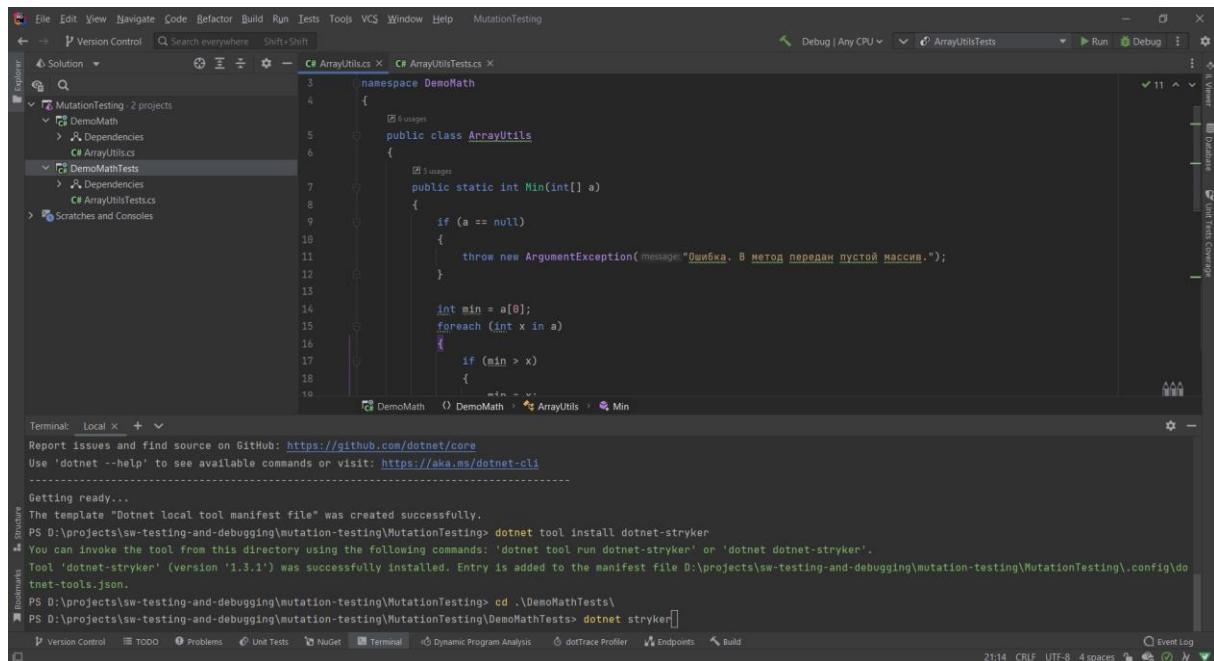


Рисунок 12.4 – Запуск мутационного тестирования

Stryker.NET автоматически определит проект для произведения мутаций основываясь на ссылках в тестовом проекте (рисунки 12.5 и 12.6).

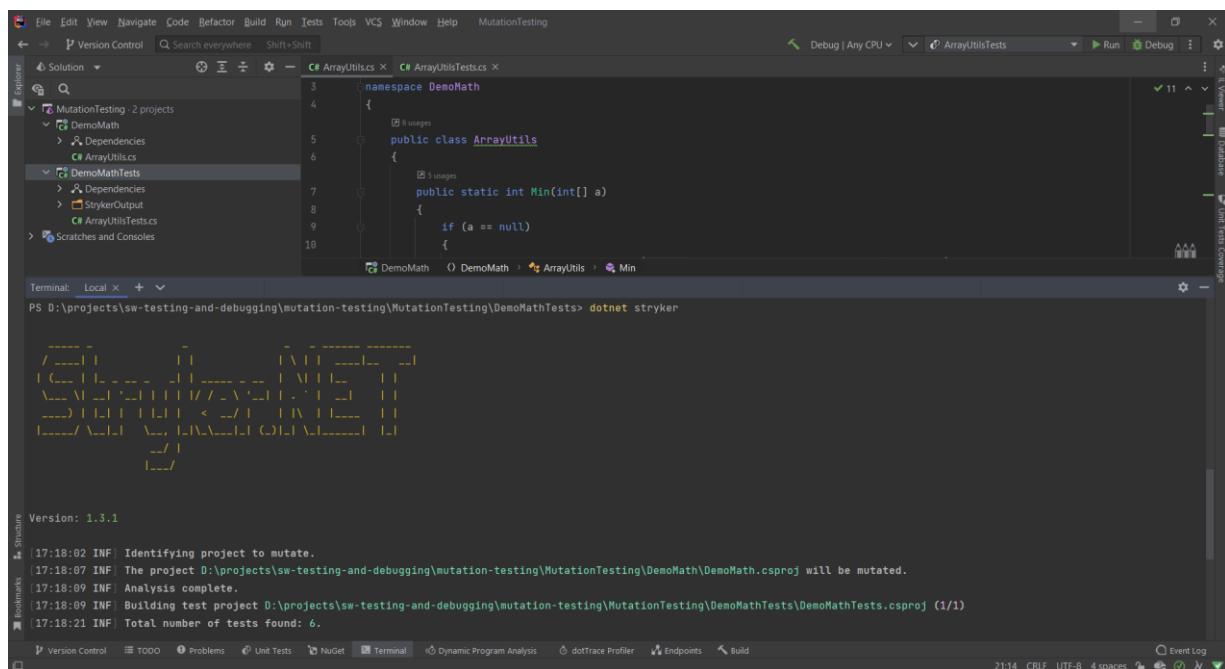


Рисунок 12.5 – Запуск Stryker.NET

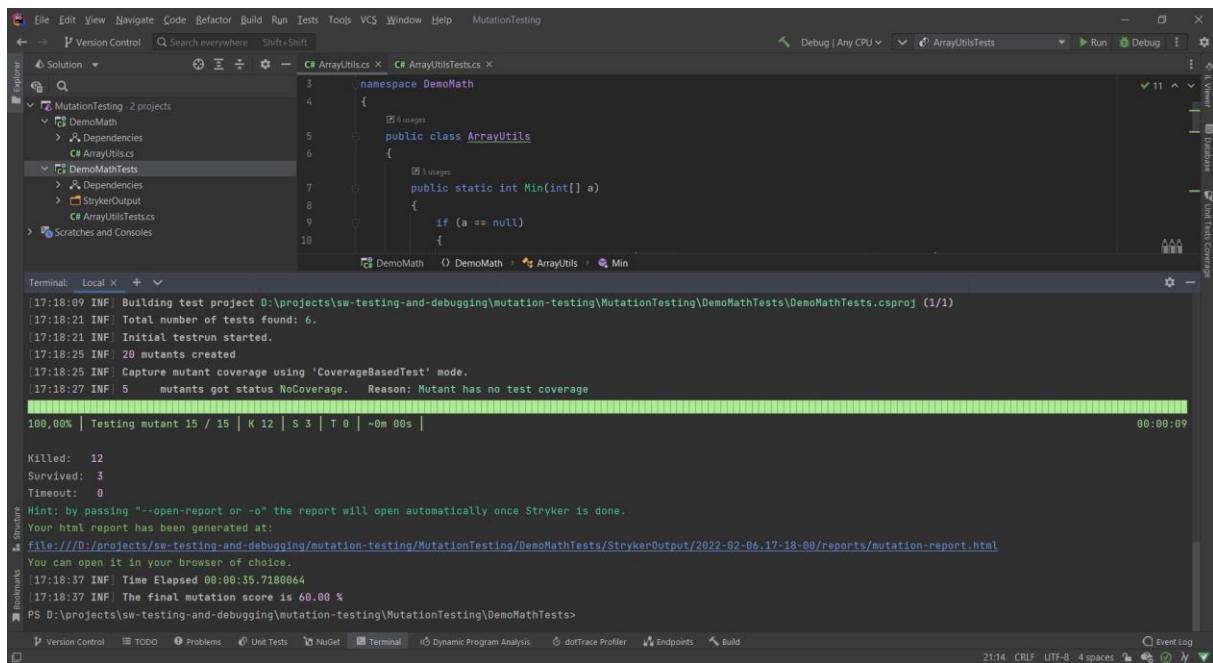


Рисунок 12.6 – Выполнение запуска Stryker.NET

На рисунке 12.6 показана базовая информация о результатах запуска. Здесь указано имя проекта (*DemoMathMath.csproj*) и то, что муттирование произошло успешно, так же указано количество сгенерированных мутантов и количество выживших. Так было сгенерировано 15 мутантов 3 из которых выжило. Однако основная информация находится в сгенерированном в формате HTML отчете (*mutation-report.html*).

После запуска будет сгенерирован отчет, на основе которого можно будет сделать вывод о качестве тестового набора (рисунок 12.7). В нем можно увидеть информацию обо всех мутированных файлах с подробностями, такими как мутационная оценка (mutation score) (больше лучше), количество уничтоженных и выживших мутантов, превышениях лимитов времени (timeouts), отсутствие покрытия, неиспользуемого кода (ignored code), а также ошибок выполнения и компиляции.

All files - Stryker.NET Report

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
All files	60.00% 60.00	12	3	0	5	0	0	0	12	8	20
ArrayUtils.cs	60.00% 60.00	12	3	0	5	0	0	0	12	8	20

Рисунок 12.7 – Отчет о запуске Stryker.NET

Мутационная оценка может варьироваться от 0 до 100. На основе таблицы можно увидеть, что из 20 сгенерированных мутантов выжило 12. Также видно, что для 5 мутантов отсутствует какое либо покрытие кода частей кода. Больше деталей можно увидеть в отчете по конкретному файлу *ArrayUtils.cs* (рисунок 12.8).

ArrayUtils.cs - Stryker.NET Report

File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected	Total mutants
ArrayUtils.cs	60.00% 60.00	12	3	0	5	0	0	0	12	8	20
< > □	<input checked="" type="checkbox"/> Killed (12) <input checked="" type="checkbox"/> Survived (3) <input checked="" type="checkbox"/> NoCoverage (5)										
1	using System;										
2											
3	namespace DemoMath										
4	{										

Рисунок 12.8 – Детальный отчет о файле

Развёрнутое представление отображает исходный код оригинального файла со всем мутациями. Выжившие мутанты отмечаются красными кружками и показывают детали при нажатии на эти кружки (рисунок 12.9).

The screenshot shows a browser window with several tabs at the top: 'Лаборатория', 'Сайт', 'РЭДПО', 'Тесты', 'Mutation', 'GitHub', 'Stryker', 'Getting Started', and 'ArrayUtil'. The main content area displays a Java code snippet for a 'Min' method:

```
6     {
7         public static int Min(int[] a)
8         {
9             if (a == null)
10            {
11                throw new ArgumentException("Ошибка. В метод передан пустой массив.");
12            }
13
14            int min = a[0];
15            foreach (int x in a)
16            {
17 -             if (min > x) ●
18 +             if (min >= x)
19             {
20                 min = x;
21             }
22         }
23     }
```

Below the code, there are three buttons: 'Killed (12)', 'Survived (3)', and 'NoCoverage (5)'. A yellow sun icon is in the top right corner. At the bottom, it says '7 🌱 Equality mutation Survived (17:21)'.

Рисунок 12.9 – Подробности реализации выжившего мутанта

Например, выживший мутант говорит, что в наборе отсутствует тест с проверкой различий операторов сравнения. Это неважно с точки зрения логики работы данного варианта реализации, так отличие будет только в количестве присваиваний переменной `min`, и данное срабатывание, по сути, является ложным.

Однако в других случаях выживший мутант может являться действительно индикатором отсутствия необходимых тестов.

Рассмотрим еще один демонстрационный пример. Пускай у нас имеется функция считающая сумму двух чисел (рисунок 12.10) и только один тест проверяющий результат (рисунок 12.11).

Листинг 12.1 – Пример метода сложения двух чисел

```
public static int Sum(int a, int b)
{
    return a + b;
}
```

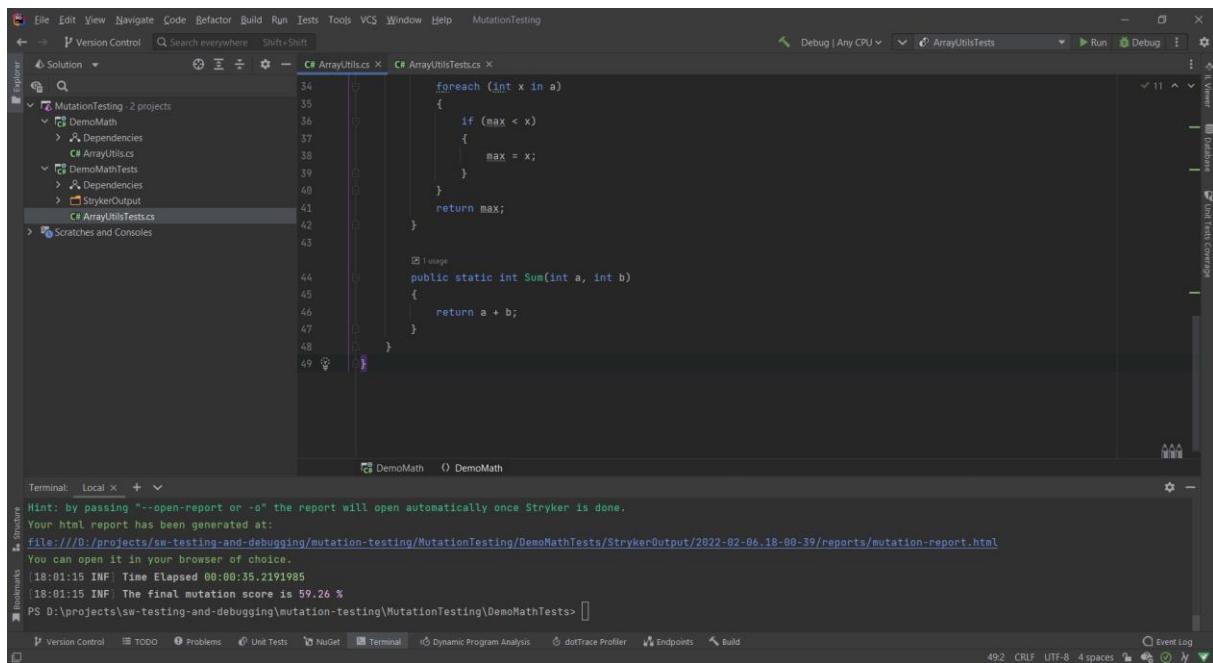


Рисунок 12.10 – Демонстрационный пример

Листинг 12.2 – Пример теста метода сложения двух чисел

```
[Test]
public void SumTest()
{
    int a = 1;
    int b = 0;
    int expected = 1;
    int actual = ArrayUtils.Sum(a, b);
    Assert.AreEqual(expected, actual);
}
```

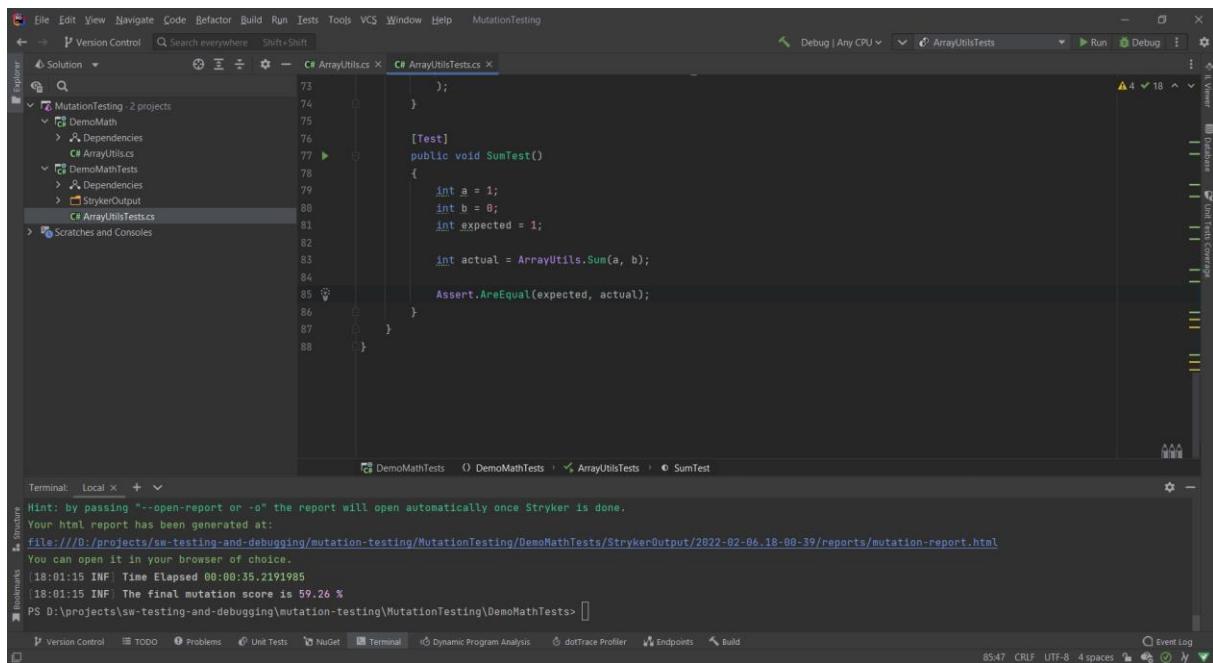


Рисунок 12.11 – Тест для демонстрационного примера

При повторном запуске мы увидим, что тестов для данного примера явно недостаточно (рисунок 12.12). Библиотекой был создан один мутант, который выжил.

The screenshot shows the Stryker UI interface. At the top, there are three buttons: **Killed (13)**, **Survived (4)**, and **NoCoverage (5)**. Below them is the mutated code for the `Sum` method:

```
35
36     if (max < x)
37     {
38         max = x;
39     }
40
41     return max;
42 }
43
44     public static int Sum(int a, int b)
45     {
46 -         return a + b;
47 +         return a - b;
48     }
49 }
```

Below the code, a message says: **21 🐚 Arithmetic mutation Survived (46:20)**.

Рисунок 12.12 – Результаты запуска мутаций для демонстрационного примера

Так например у нас отсутствует тест проверяющий мутацию с изменением знака плюс на минус. Этот мутант не был уничтожен, так как созданный нам тест не учитывает данную возможность, и данные для него были подобраны не удачно.

```
Original
var actual = a + b;
var actual = 1 + 0;
var actual = 1;

Mutated
var actual = a - b;
var actual = 1 - 0;
var actual = 1;
```

Для выбранных входных данных ($a = 1$, $b = 0$) и оригинальная и мутированная версии метода возвращают один и тот же результат, который ожидает модульный тест. Это четко говорит нам о том, что покрытие данного метода не достаточно хорошее. Для исправления ситуации необходимо улучшить тест, взяв другие входные значения.

Листинг 12.3 – Пример улучшенного теста для метода сложения двух чисел

```
[Test]
public void SumTest()
{
    int a = 1;
    int b = 2;
    int expected = 3;
    int actual = ArrayUtils.Sum(a, b);
    Assert.AreEqual(expected, actual);
}
```

В этот раз был создан так же один мутант, который был успешно уничтожен благодаря изменениям в teste (рисунок 12.13).

```
34     foreach (int x in a)
35     {
36         if (max < x)
37         {
38             max = x;
39         }
40     }
41     return max;
42 }
43
44 public static int sum(int a, int b)
45 {
46     return a + b;
47 }
48 }
49 }
```

Рисунок 12.13 – Изменения после обновления теста

3 Контрольные вопросы

- 4) Что представляет собой мутационное тестирование?
- 5) Что такое мутант?
- 6) С какой целью применяется мутационное тестирование?
- 7) Какие виды мутаций бывают?
- 8) Какие недостатки у мутационного тестирования?
- 9) Что представляет собой библиотека Stryker.NET?

4 Задание

- 4) Проанализировать качество тестов в проектах из предыдущих лабораторных работ.
- 5) Расширить тестовые наборы для обнаружения всех мутантов.
- 6) Оформить отчёт.

Лабораторная работа № 12. Тестирование программ при отсутствии исходного кода методом черного ящика

1 Цель работы

Закрепить принципы тестирования методом черного ящика и использования внешних библиотек.

2 Краткая теория

Иногда возникает ситуация, когда исходный код библиотеки не доступен (например, был случайно удален или потерян), но имеется сборка с важной функциональностью и необходимо воссоздать исходный код с идентичным поведением. В этом случае необходимо зафиксировать существующее поведение в виде набора тестов для того, что бы иметь возможность опираться на них во время повторения функциональности. Единственным вариантом в этом случае является применение метода черного ящика для написания тестов.

2.1 Подключение внешней библиотеки в существующий проект

Перед началом работы необходимо скопировать подключаемую библиотеку в каталог с проектом (рисунок 12.1). Это необходимо для того что бы было возможно указать относительный путь к библиотеке.

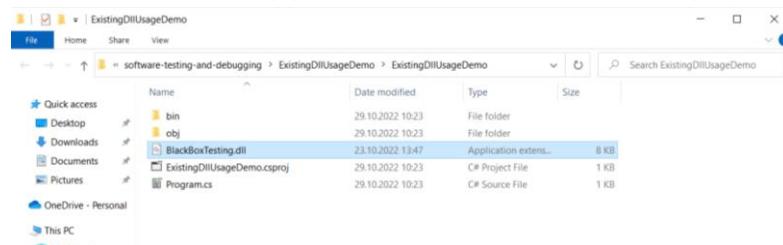


Рисунок 12.1 – Размещение библиотеки в каталоге приложения

2.1.1 Подключение внешней библиотеки в существующий проект в Visual Studio 2017/2019/2022

Для подключения существующей библиотеки необходимо добавить ее в список зависимостей (рисунок 12.2) и выбрать Browse в открывшемся окне (рисунки 12.3 - 12.5).

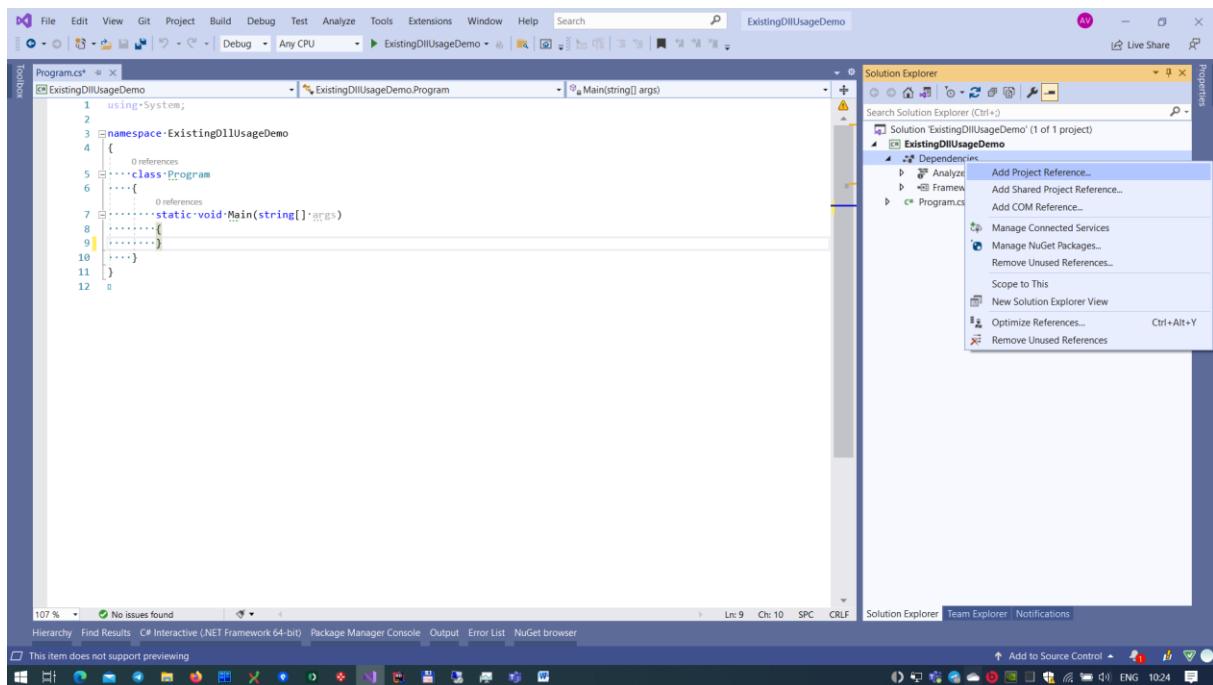


Рисунок 12.2 – Добавление библиотеки в список зависимостей

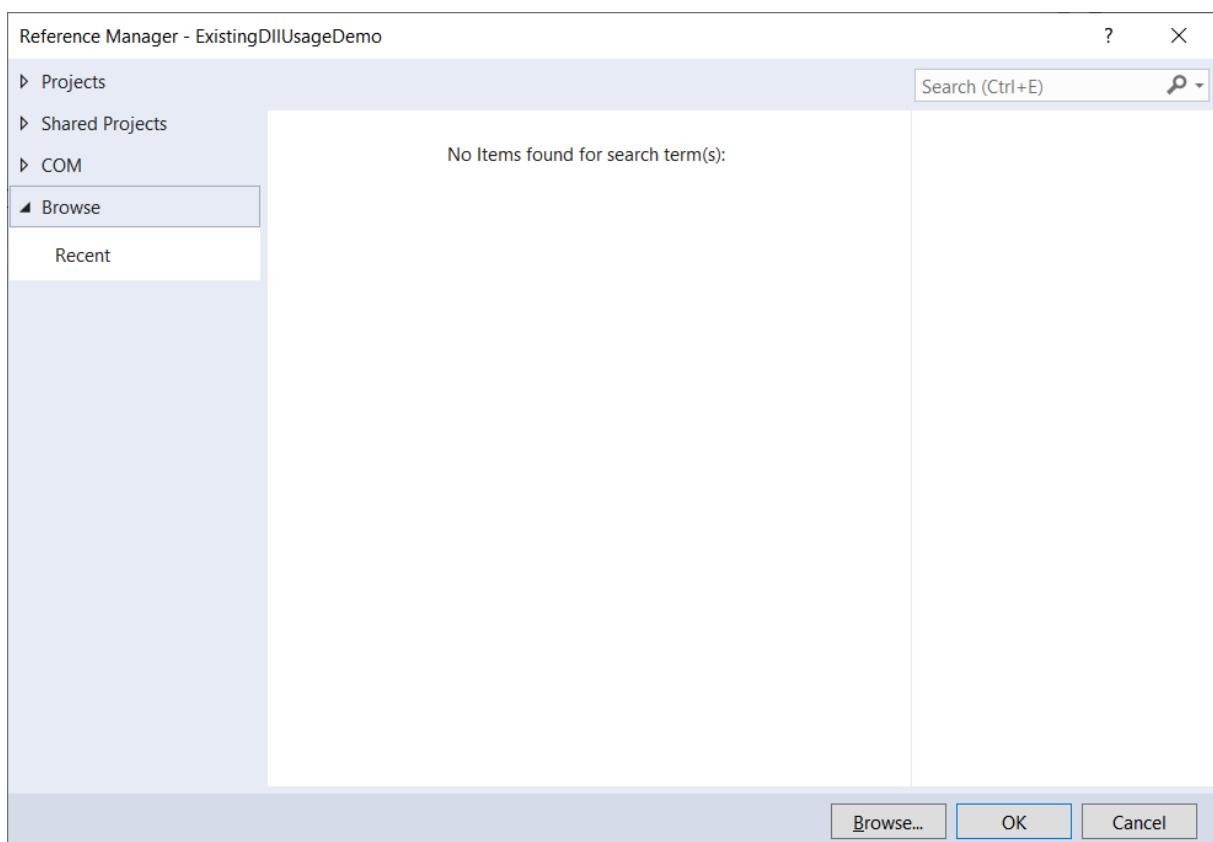


Рисунок 12.3 – Выбор Browse для указания пути к библиотеке

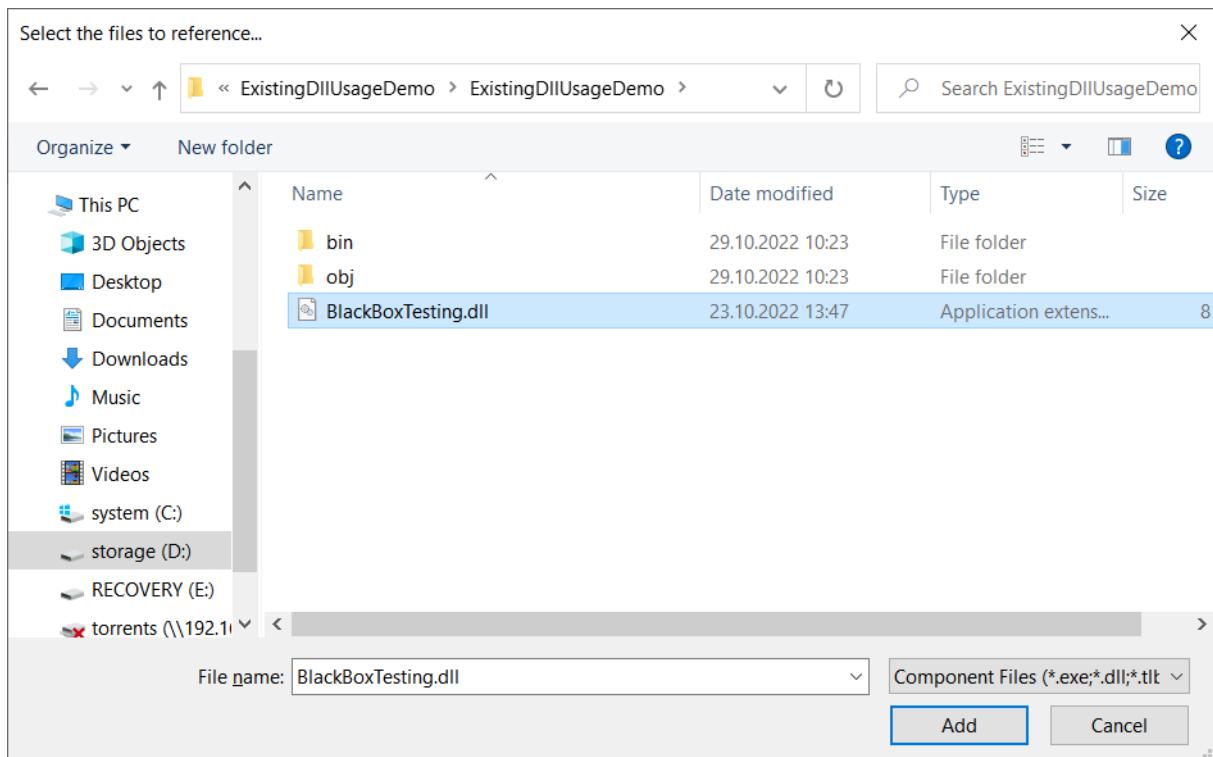


Рисунок 12.4 – Указание пути к библиотеке

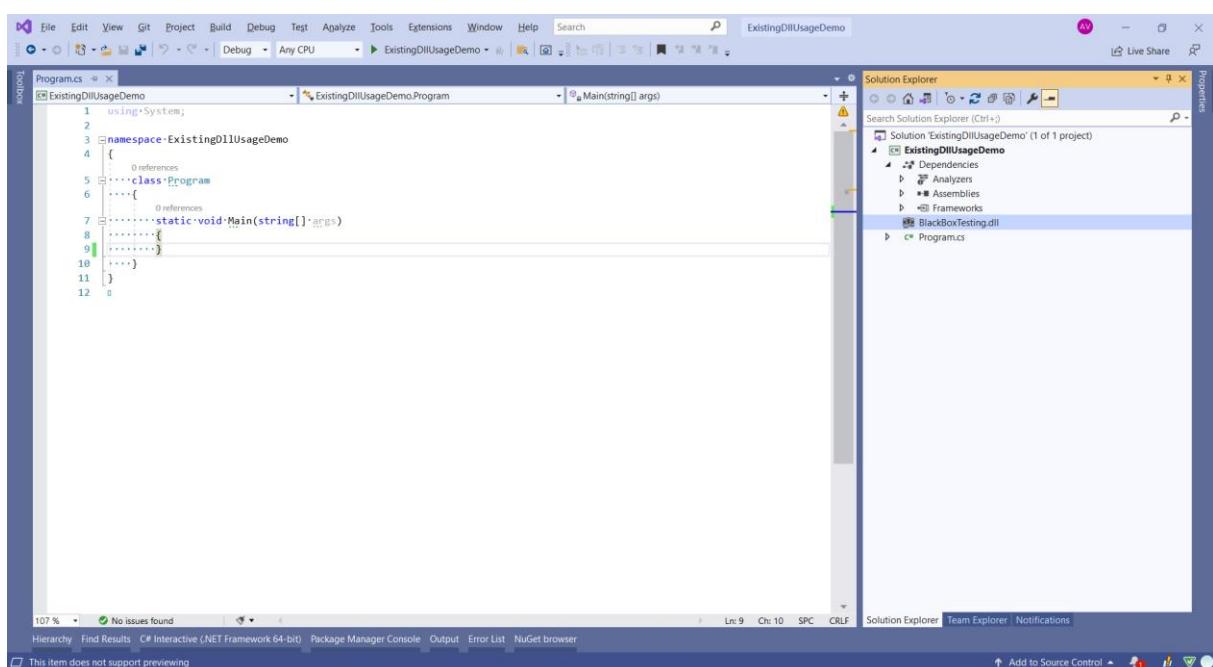


Рисунок 12.5 – Результат добавления библиотеки в список зависимостей

Далее для использования кода библиотеки необходимо указать соответствующее пространство имен (рисунок 12.6)

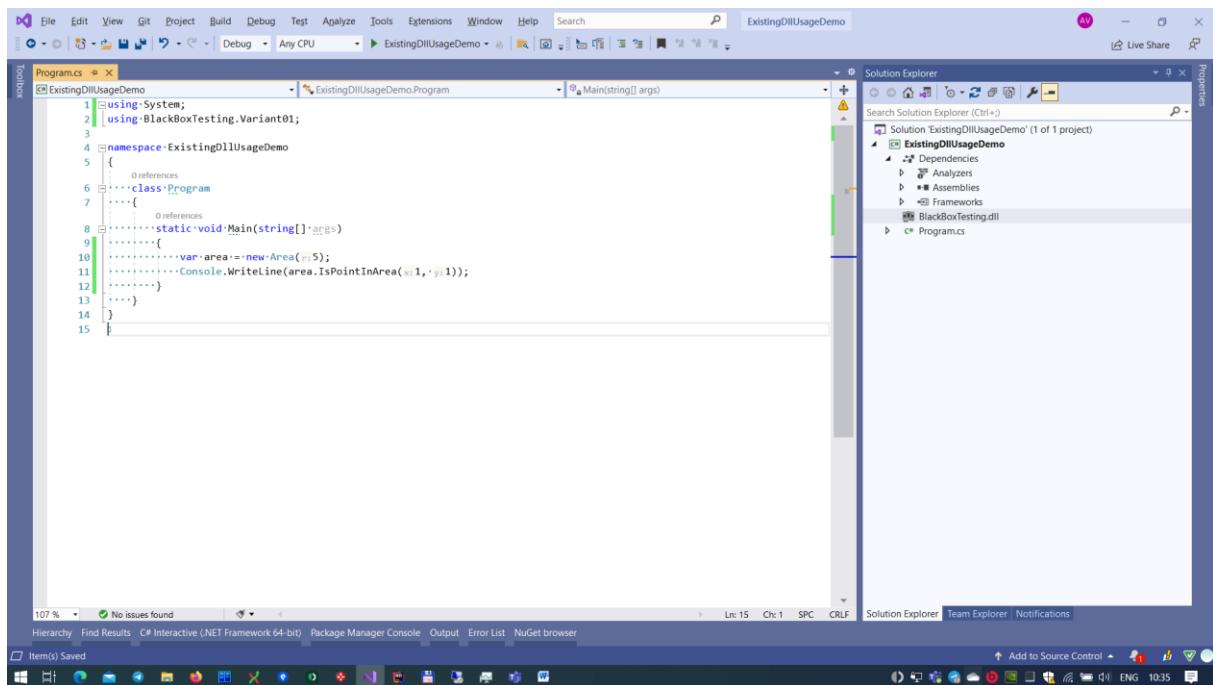


Рисунок 12.6 – Использование кода библиотеки с помощью подключения пространства имен

2.1.2 Подключение внешней библиотеки в существующий проект в JetBrains Rider 2022

Для подключения существующей библиотеки необходимо добавить ее в список зависимостей (рисунок 12.2) и выбрать Add From в открывшемся окне (рисунки 12.3 - 12.5).

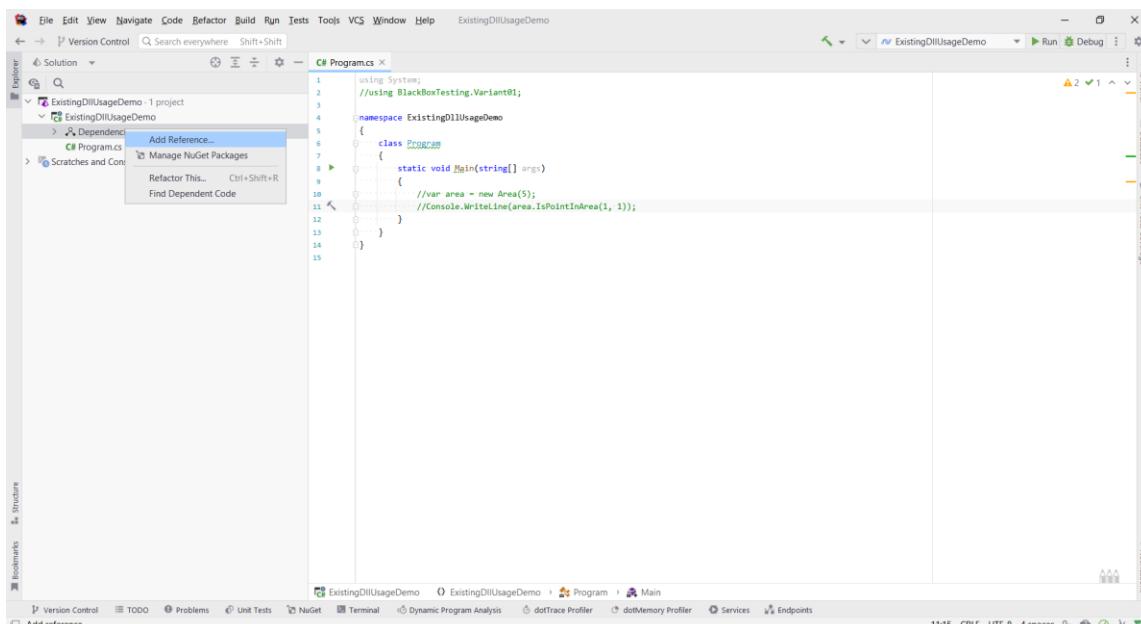


Рисунок 12.2 – Добавление библиотеки в список зависимостей

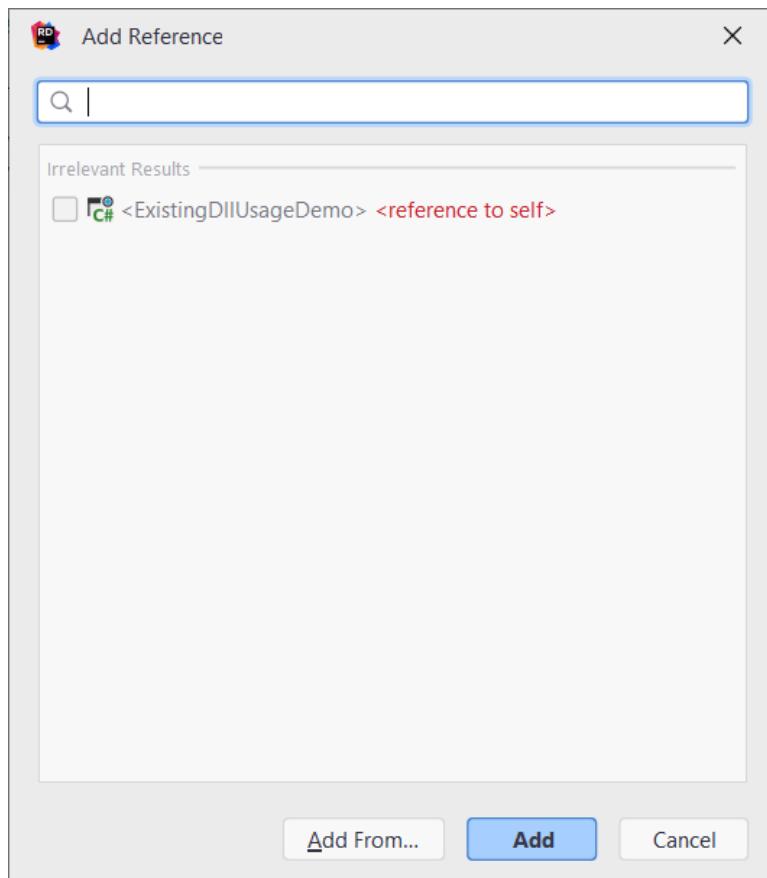


Рисунок 12.3 – Выбор Add From для указания пути к библиотеке

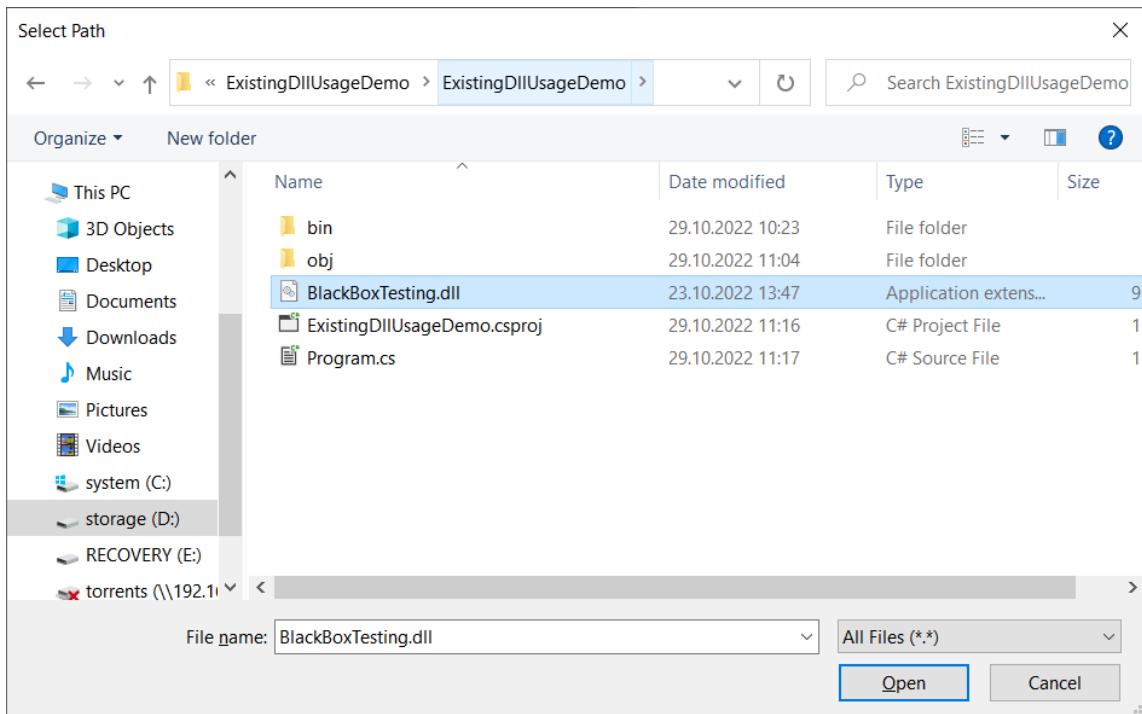


Рисунок 12.4 – Указание пути к библиотеке

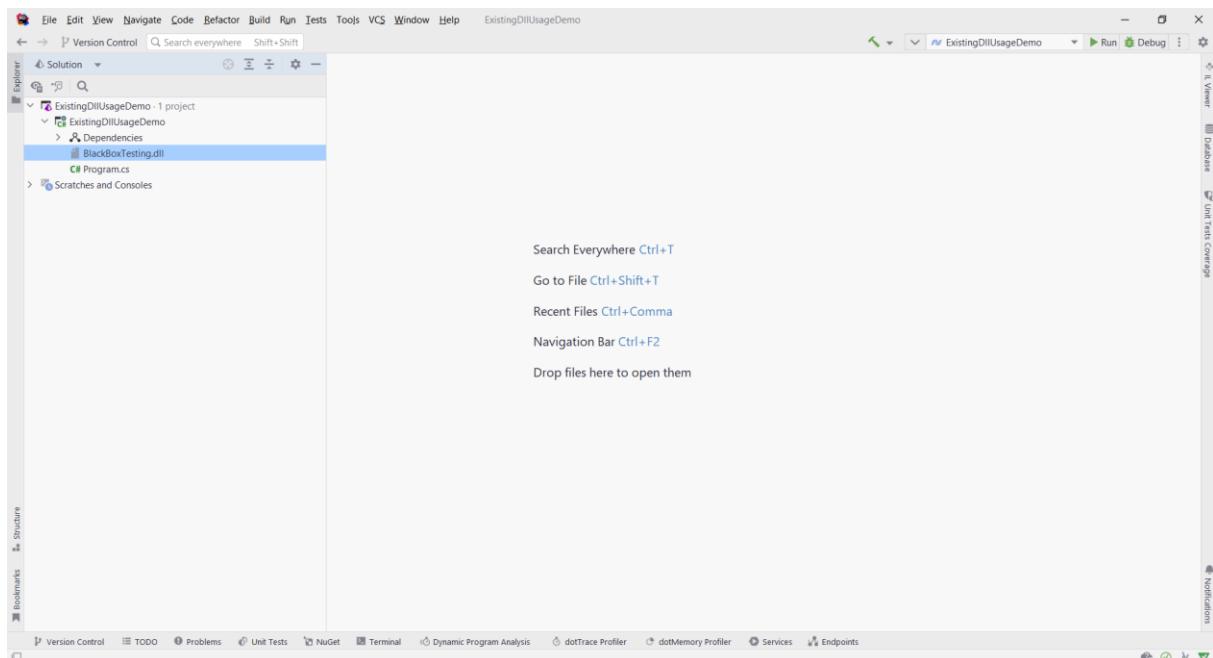


Рисунок 12.5 – Результат добавления библиотеки в список зависимостей

Далее для использования кода библиотеки необходимо указать соответствующее пространство имен (рисунок 12.6)

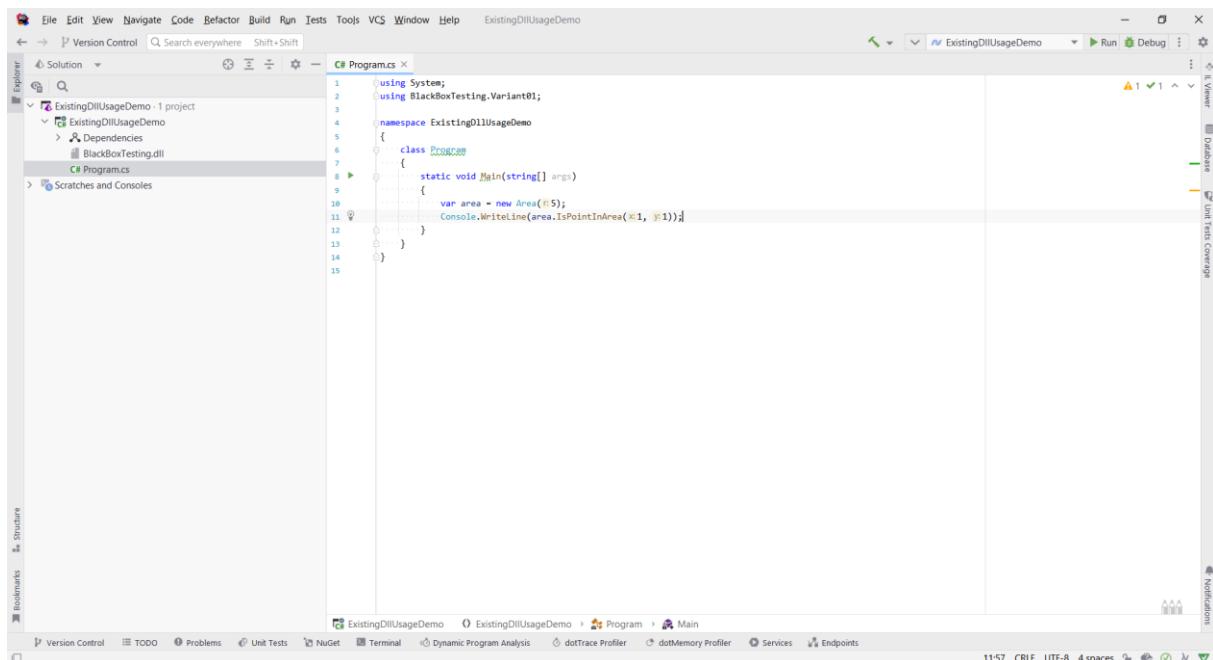


Рисунок 12.6 – Использование кода библиотеки с помощью подключения пространства имен

3 Контрольные вопросы

- 1) Как подключить внешнюю библиотеку к проекту?
- 2) Как использовать код внешней библиотеки в проекте?

4 Задание

- 1) Подключить внешнюю библиотеку к новому проекту.
- 2) В код части классов библиотеки целенаправленно были внесены одна или несколько ошибок. Часть классов работает корректно.
- 3) Создать проект с модульными тестами для подключенной библиотеки и написать тесты для класса Area реализующей проверку попадания в закрашенную область фигуры в соответствии с вариантом задания из пункта 5. Необходимо обнаружить ошибки в коде (если они есть) и указать при каких условиях они возникают.
- 4) Оформить отчёт.

5 Варианты заданий

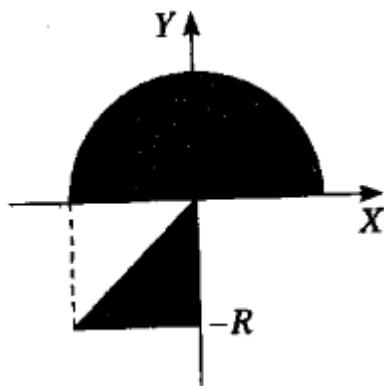
Подключаемые классы Area (реализующие общий интерфейс IArea) находятся в соответствующих пространствах имен (Variant01 – для варианта 1, Variant02 – для варианта 2, и т.д.).

Значения, характеризующие фигуры (R , a , b , и т.д.) задаются через конструктор.

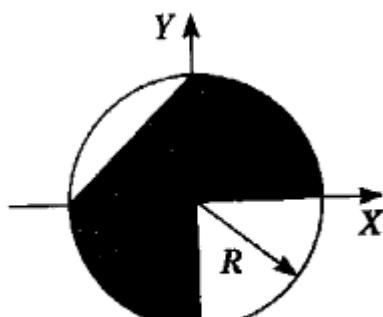
В составе класса имеется метод IsPointInArea, определяющий принадлежность точки с заданными координатами хотя бы одной из закрашенных областей. Сигнатура метода представлена ниже:

```
namespace BlackBoxTesting
{
    public interface IArea
    {
        bool IsPointInArea(double x, double y);
    }
}
```

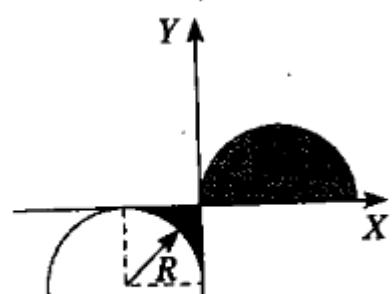
Варианты заданий:



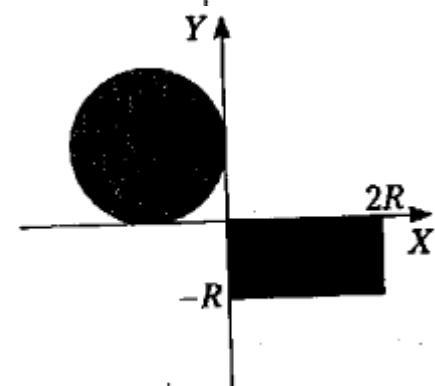
1)



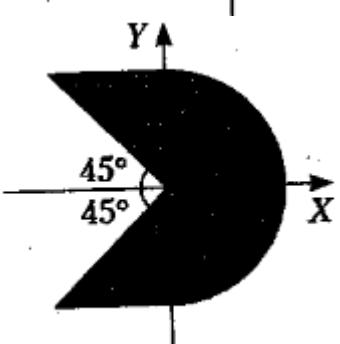
2)



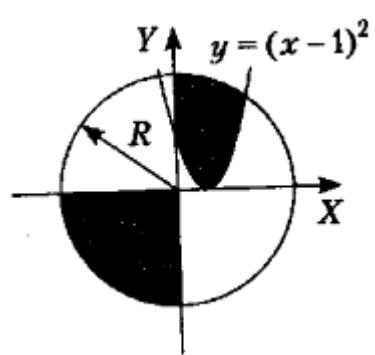
3)



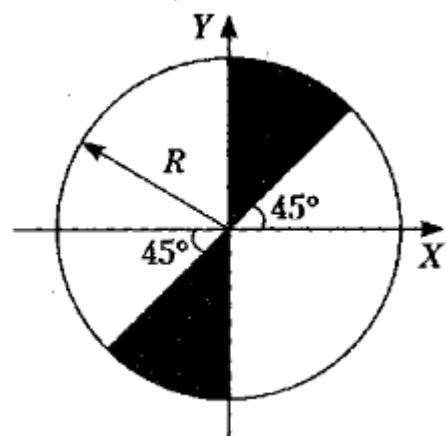
4)



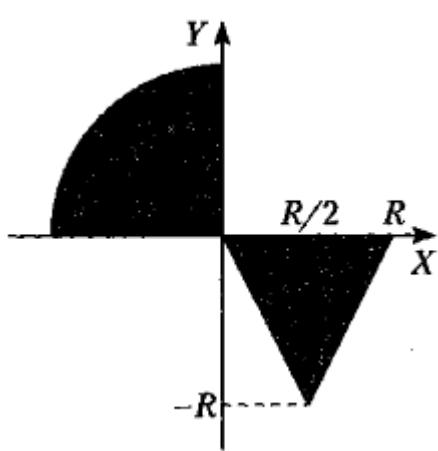
5)



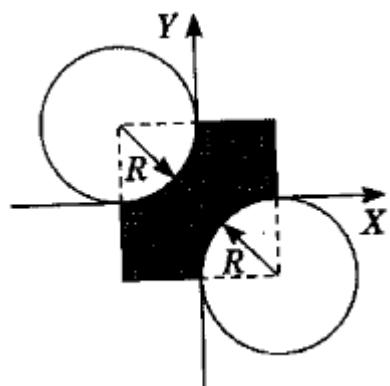
6)



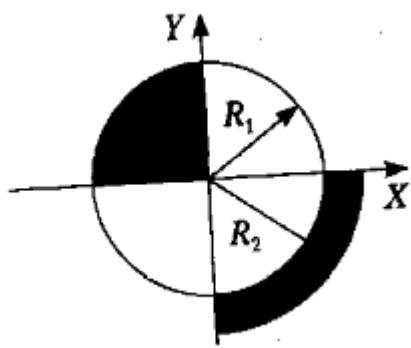
7)



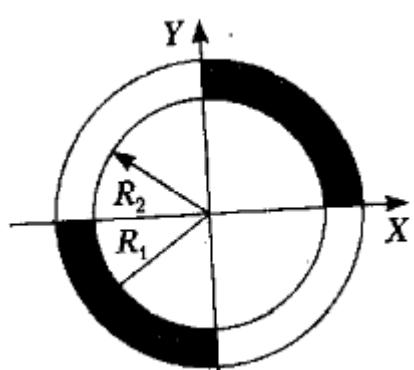
8)



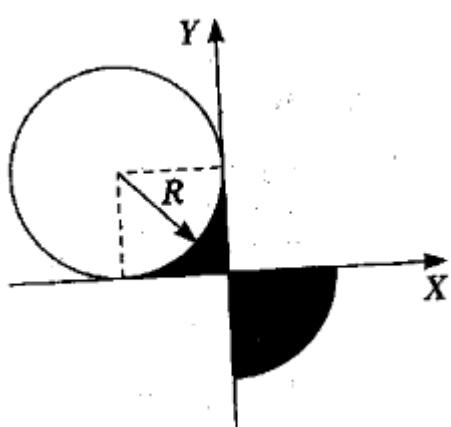
9)



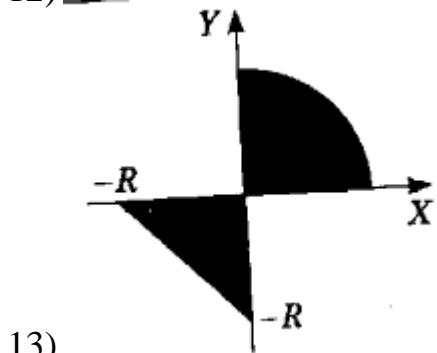
10)



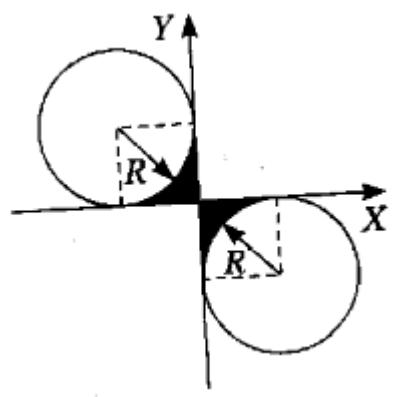
11)



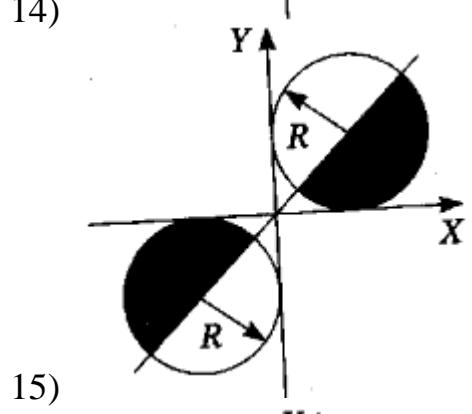
12)



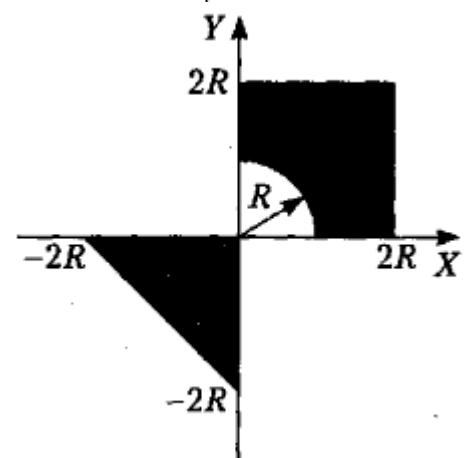
13)



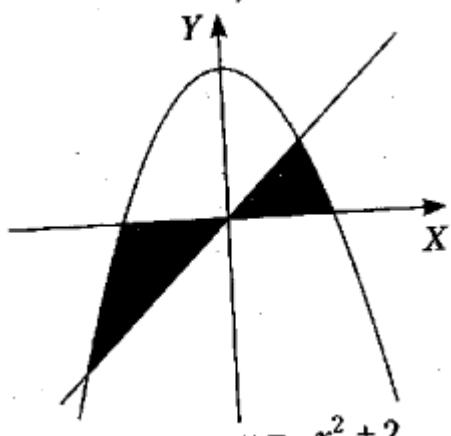
14)



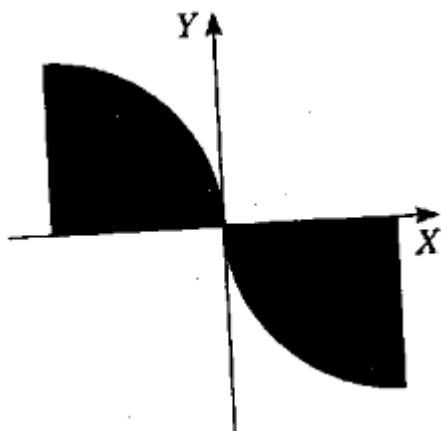
15)



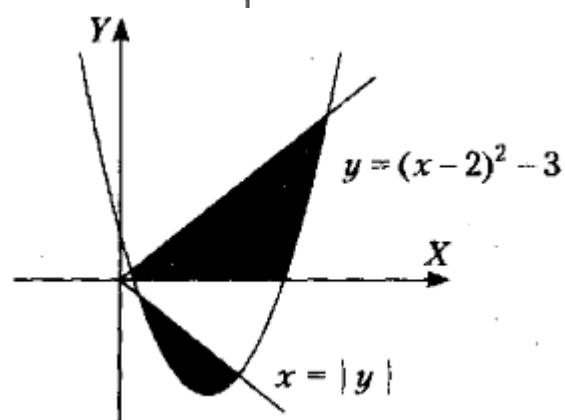
16)



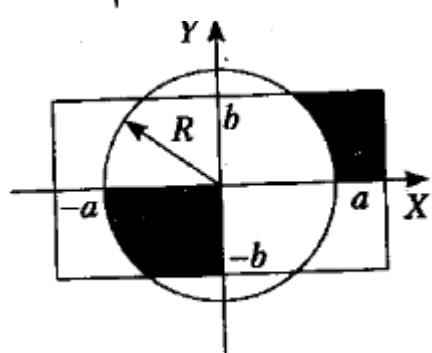
17)



18)



19)



20)

Приложение А (справочное)

Работа с системой управления репозиториями кода gitlab

1 Регистрация на gitlab.com

GitLab – веб-инструмент жизненного цикла DevOps с открытым исходным кодом, представляющий систему управления репозиториями кода для Git с собственной вики, системой отслеживания ошибок, CI/CD пайплайном и другими функциями.

Для регистрации необходимо зайти на сайт gitlab.com (рисунок А.1) и перейти к регистрации (рисунок А.2).

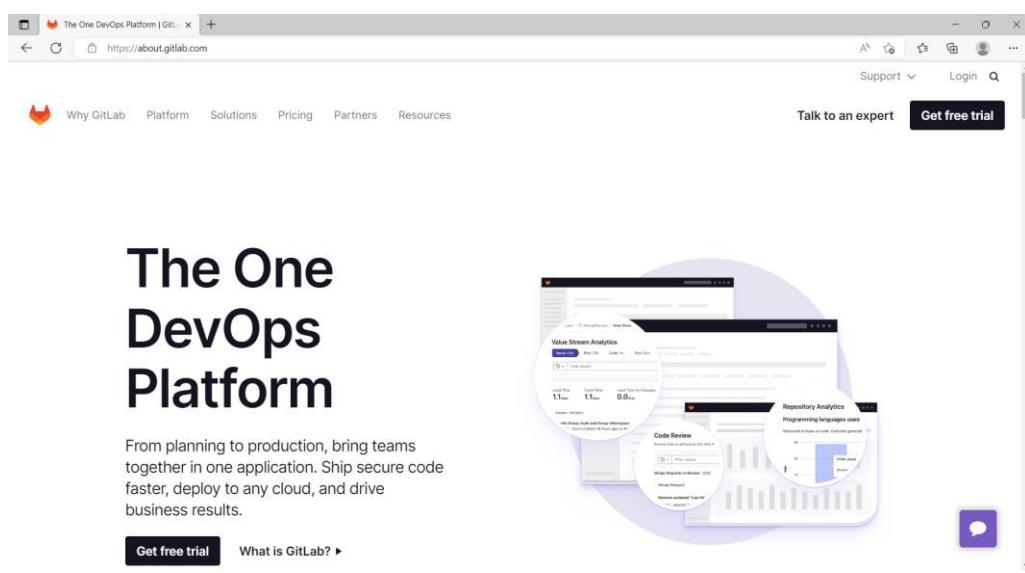


Рисунок А.1 – Главная страница gitlab.com

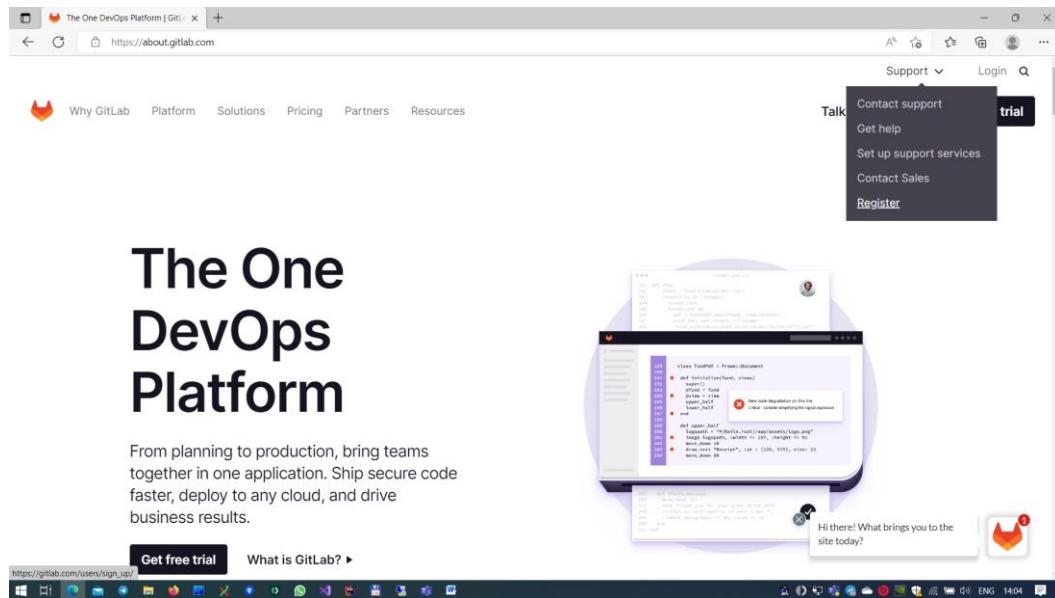


Рисунок А.2 – Переход на страницу регистрации gitlab

Далее необходимо ввести фамилию и имя (настоятельно рекомендуется, что бы они совпадали с реальными данными студента: так проще идентифицировать сдающего лабораторные работы), имя пользователя, адрес электронной почты и пароль (рисунок А.3).

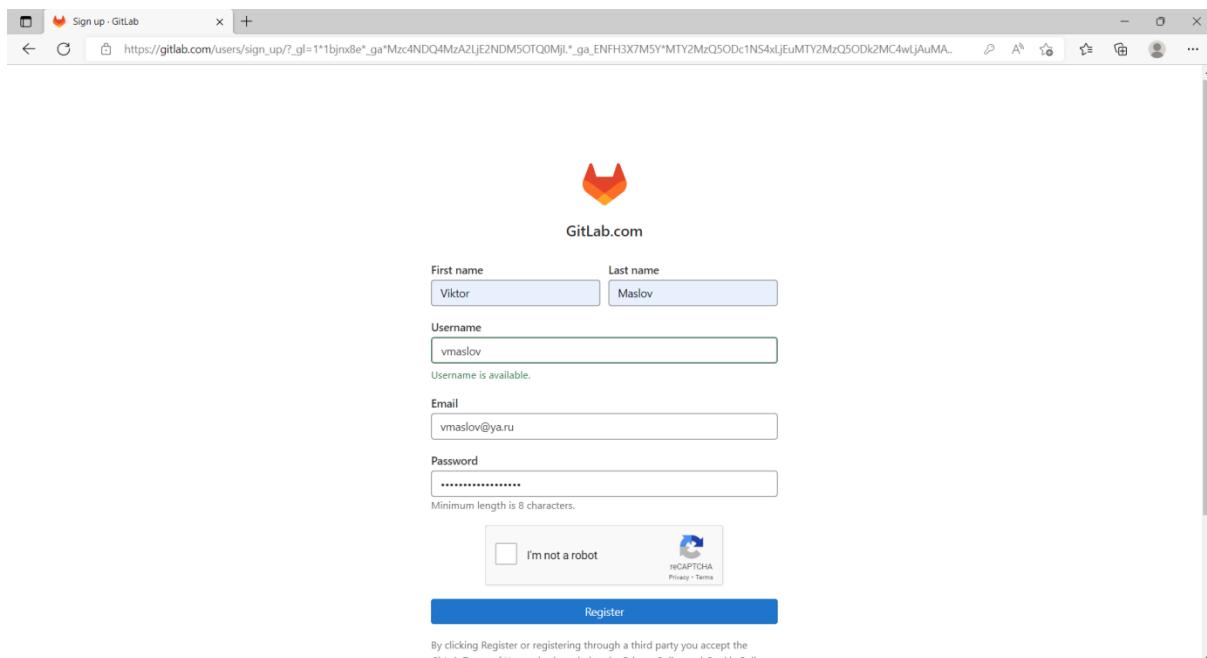


Рисунок А.3 – Страница регистрации gitlab

После регистрации необходимо завершить регистрацию, подтвердив почтовый ящик. Далее необходимо завершить регистрацию и войти в систему.

Далее необходимо отправить имя зарегистрированного пользователя преподавателю для предоставления доступа к проекту дисциплины (для примера на скриншотах это будет vmaslov).

2 Работа с проектом для лабораторных работ

После получения доступа к проекту дисциплины (рисунок А.4) (письмо с адресом проекта будет отправлено на регистрационную почту) студенту будет предоставлен доступ к 2 подгруппам (рисунок А.5):

- Labs – подгруппа с репозиториями (проектами) для сдачи лабораторных работ. Внутри данной подгруппы созданы подгруппы для учебных групп (рисунок А.6), а далее располагается персональный репозиторий для выполнения лабораторных работ (рисунок А.7).

- Materials – репозиторий (проект) с материалами дисциплины: МУ по выполнению лабораторных работ и т.д.

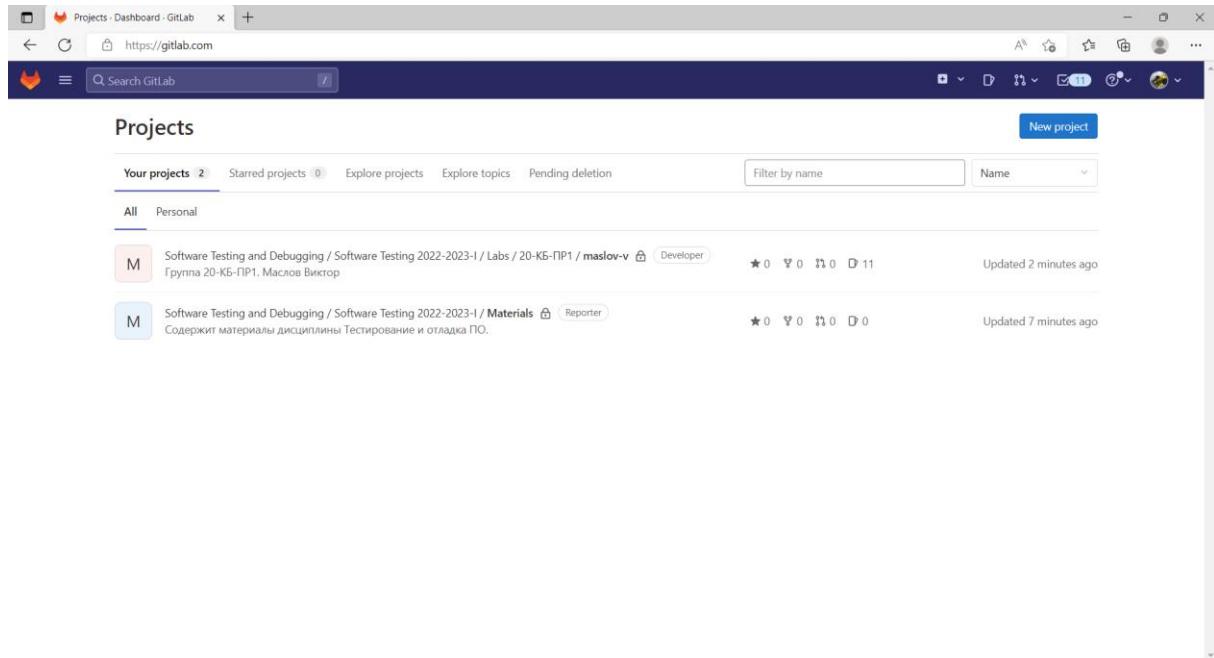


Рисунок А.4 – Главная страница после предоставления доступа к проектам дисциплины на gitlab

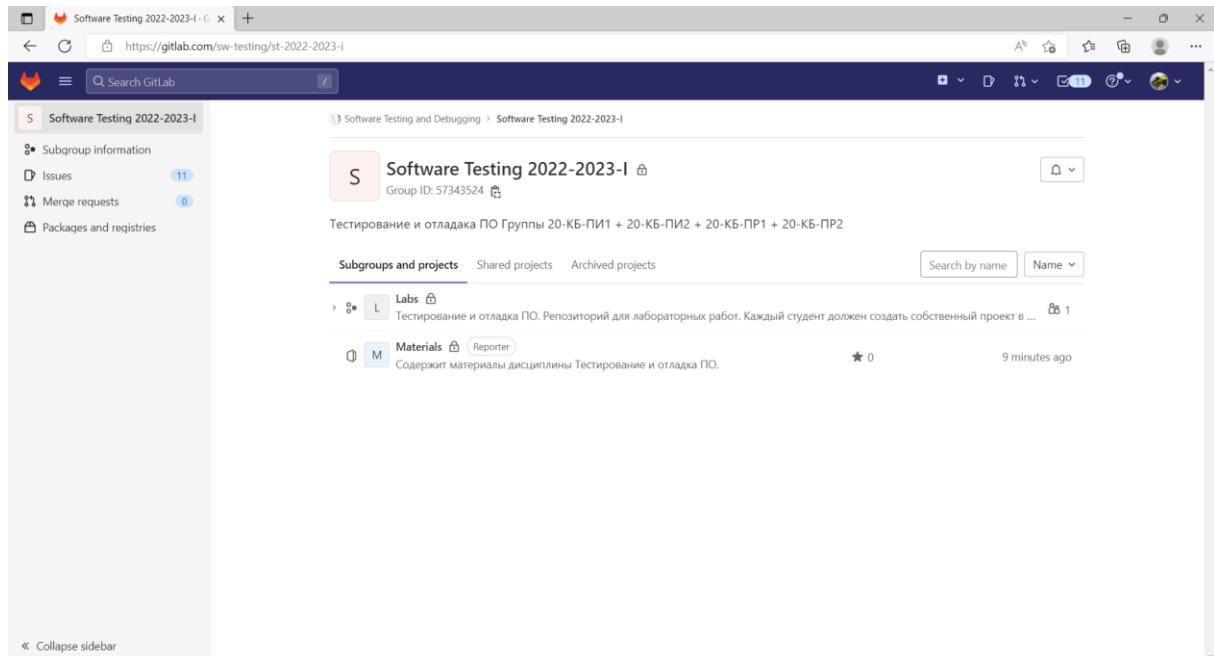


Рисунок А.5 – Проекты дисциплины на gitlab

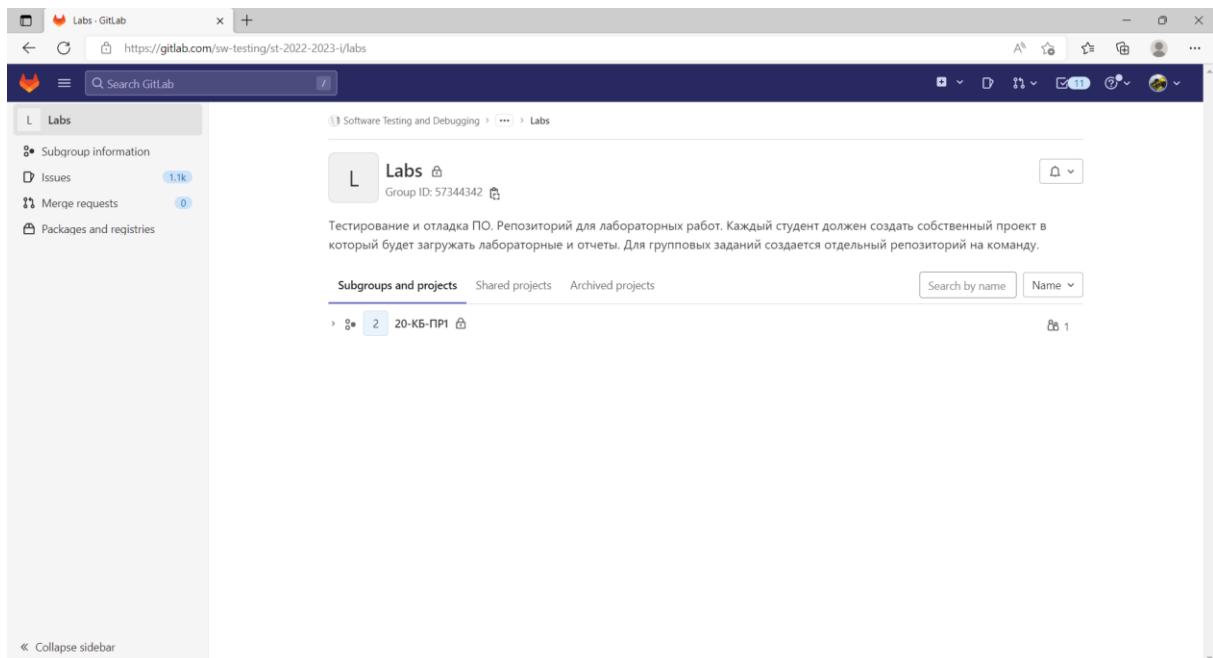


Рисунок А.6 – Подгруппа Labs с указанием для учебной группы

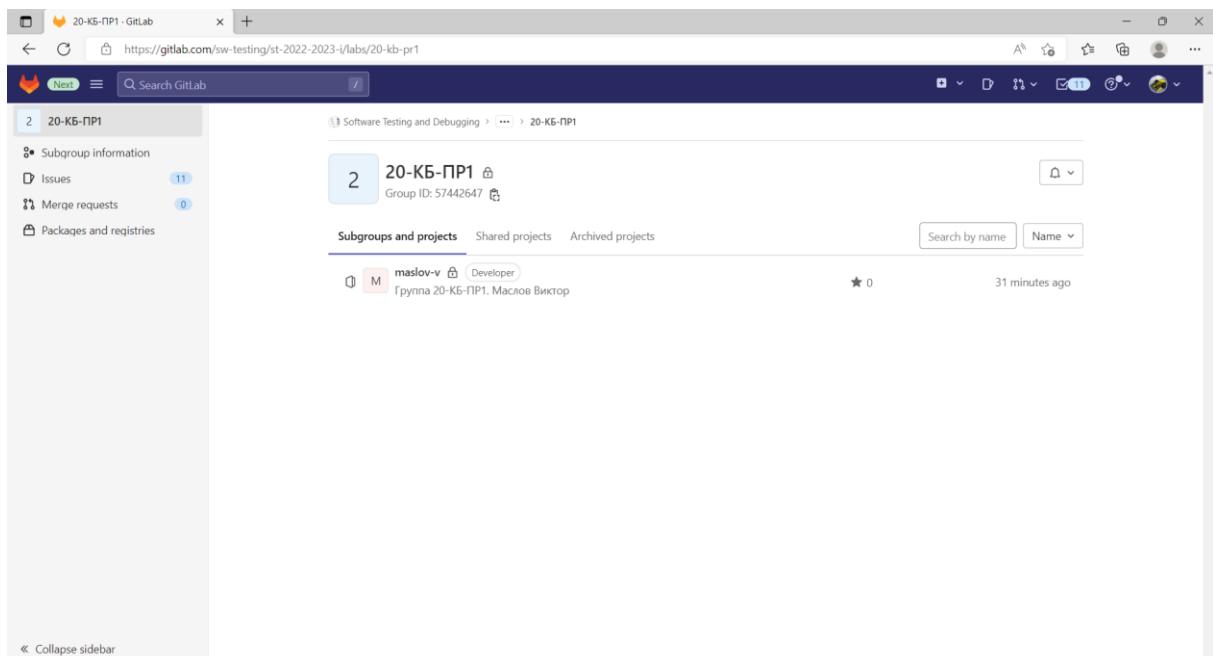


Рисунок А.7 – Персональный проект для лабораторных работ

3 Работа с проектом дисциплины

Для дальнейшей работы необходимо перейти в персональный проект с репозиторием для сдачи лабораторных работ (рисунок А.8) и выполнить клонирование на локальный компьютер (рисунок А.9).

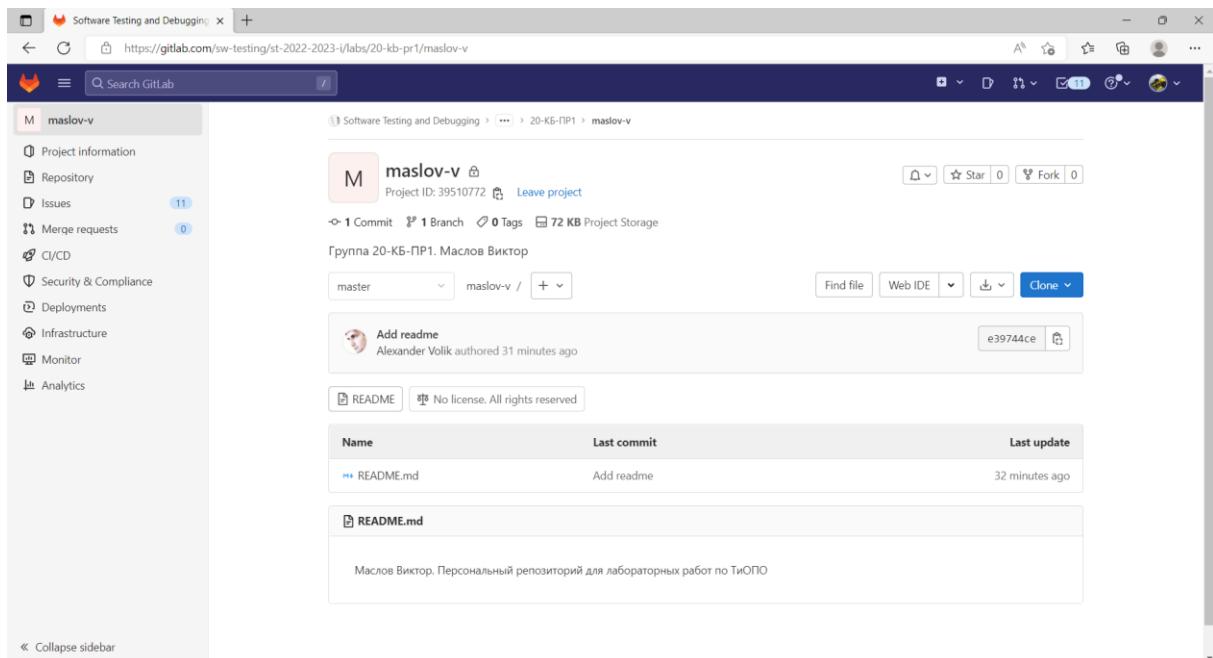


Рисунок А.8 – Основой проект дисциплины для лабораторных работ

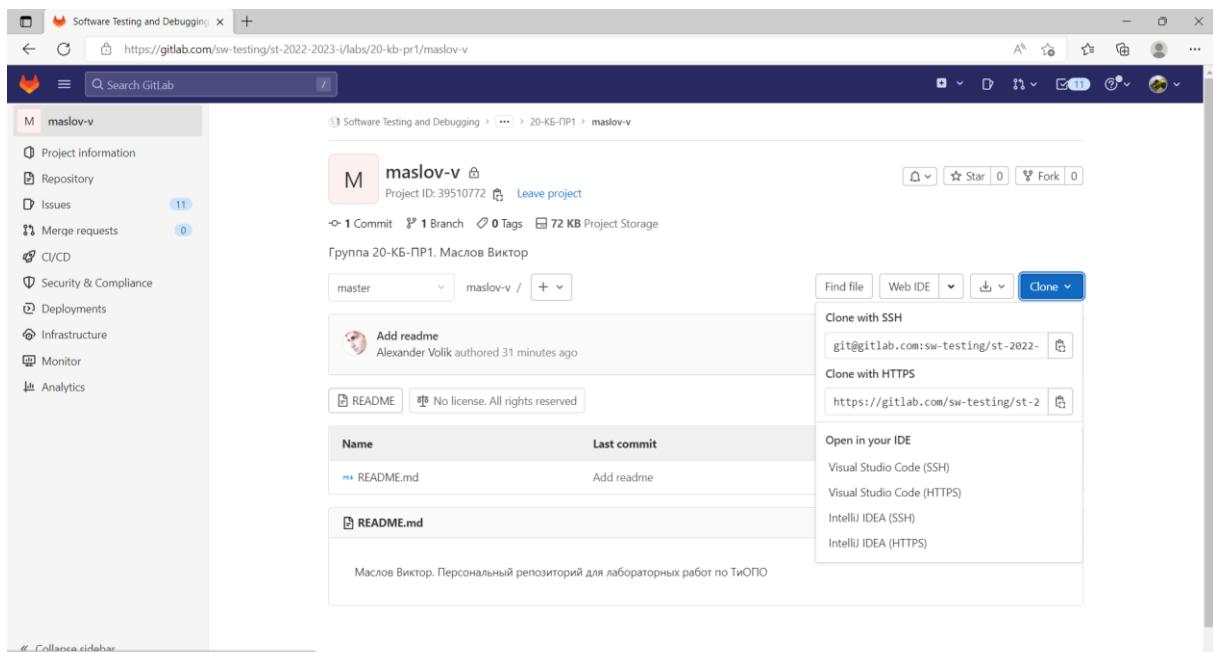


Рисунок А.9 – Клонирование нового проекта на локальный компьютер

Дальнейшая работа с репозиторием, получение и отправка изменений производится в обычном режиме.

4 Работа с системой задач

Удобным способом взаимодействия через систему gitlab является система задач (issues). Для каждой лабораторной работы создается

отдельная задача и с помощью них отслеживается прогресс выполнения лабораторных работ.

При создании проекта в него заранее были импортированы задачи, соответствующие лабораторным работам дисциплины (рисунок А.10). В каждой задаче имеется номер и название лабораторной работы (рисунок А.11)

The screenshot shows the GitLab interface for the 'Software Testing and Debugging' project under the '20-КБ-ПР1' namespace. The 'Issues' section is selected in the sidebar, showing 11 open issues. The list includes tasks such as 'STDLab11', 'STDLab10', 'STDLab09', 'STDLab08', 'STDLab07', 'STDLab06', 'STDLab05', 'STDLab04', and 'STDLab03'. Each task has a creation timestamp and an 'updated' timestamp. The interface includes standard GitLab navigation and search tools.

Рисунок А.10 – Список задач дисциплины на gitlab

The screenshot shows the detailed view of the first issue, 'STDLab01'. The issue was created by Alexander Volik 41 minutes ago. The description states: 'Лабораторная работа № 1. Отладка программ в интегрированных средах разработки'. The 'Tasks' section indicates 0 tasks assigned. The 'Linked items' section shows assignments: Alexander Volik assigned to @vmaslov 41 minutes ago, and Viktor Maslov assigned to @vmaslov just now. The right side of the screen displays various settings and status indicators for the issue, such as 'Assignee' (Viktor Maslov), 'Epic' (locked), 'Labels' (None), 'Milestone' (None), 'Due date' (None), 'Time tracking' (No estimate or time spent), 'Confidentiality' (Not confidential), and 'Lock issue' (Unlocked).

Рисунок А.11 – Задача для лабораторной работы №1

Далее по мере выполнения каждой работы необходимо назначать задачу по выполненной лабораторной задаче на преподавателя для проверки (рисунки А.12 и А.13).

The screenshot shows the GitLab interface for a project named 'STDLab01'. In the sidebar, under 'Issues', there is a red box around the 'Assignee' section. The 'Assignee' dropdown is set to 'Select assignee' and has a 'Search users' input field. Below it, 'Unassigned' is listed. A list of users includes 'Viktor Maslov @vmaslov' (selected), 'Alexander Volik @volikag', and 'Akulasj'. Other sections visible include 'Tasks' (0), 'Linked items' (0), and a note about dragging designs or clicking to upload. The right side of the interface shows various settings like 'Time tracking', 'Confidentiality', and 'Lock issue'.

Рисунок А.12 – Назначение задачи для проверки на преподавателя

This screenshot is identical to Figure A.12, showing the assignment of a task to 'Alexander Volik'. However, the 'Assignee' dropdown now shows 'Alexander Volik' selected, indicating the task has been assigned. The rest of the interface, including the sidebar and right-hand settings, remains the same.

Рисунок А.13 – Задача назначена для проверки на преподавателя

В случае успешного выполнения и проверки, преподаватель закрывает задачу с соответствующим комментарием (рисунок А.14). В

противном случае в комментариях будут указаны замечания, которые необходимо устранить и проторить процесс назначения задачи.

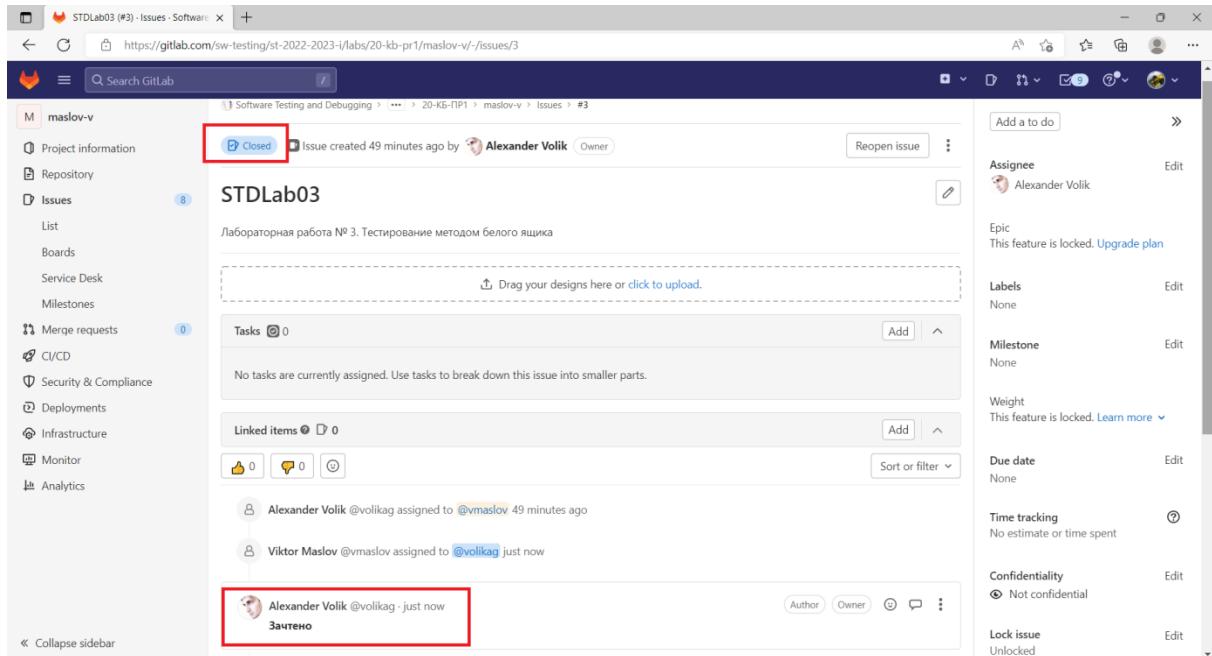


Рисунок А.14 – Результат проверки работы

4 Использование Merge Requests при выполнении лабораторных работ

Merge Request (мердж-реквест) – это запрос на слияние одной ветки в другую. Чаще всего вливается текущая рабочая ветка в master (или main) – основную ветку разработки. В первую очередь Merge Requests нужны при работе в команде для проведения код-ревью, чтобы коллеги увидели новый код и как минимум его просмотрели и оставили замечания или вопросы. Мы будем использовать их для контроля качества кода и проверки корректности выполнения лабораторных работ.

В github и bitbucket используется понятие pull request, в gitlab – merge request, хотя с точки зрения назначения и использования они эквивалентны.

В общем виде процедура мердж-реквеста обычно проходит следующим образом:

1. Сначала готовится ветка с новым функционалом/исправлением бага/рефакторингом.
2. Ветка заливается на gitlab.

3. В интерфейсе gitlab создается мердж-реквест, выбирается, какую ветку и куда необходимо смерджить (чаще всего в мастер), при необходимости указываем описание мердж-реквеста и ревьюеров (тех, кто будет смотреть наш код).

4. Коллеги получают оповещение о новом мердж-реквесте.

5. Коллеги смотрят код, оставляют замечания или вопросы.

6. Далее происходит обсуждение и анализ замечаний: можно соглашаться и выполнить исправления или возражать, объясняя, почему было сделано именно так.

7. Выполняются правки и производится заливка новых коммитов в ту же ветку.

8. Авторы замечаний разрешают все вопросы, убеждаясь что все замечания были закрыты или поправлены.

9. Мердж-реквест принимается, при этом ветка слиивается в мастер.

4.1 Создание Merge Requests через задачи (issues)

Одним из удобных способов создания мердж-реквестов является использование задач (issue). Основным плюсом такого использования является то, что задача автоматически связывается с мердж-реквестом и закрывается после его принятия.

Для того чтобы создать новый мердж-реквест для задачи необходимо перейти в нее и выбрать создание мердж-реквеста из выпадающего меню (рисунок А.15). При этом автоматически будет создана отдельная ветка для выполнения задачи (можно исправить ее имя в соответствующем поле). После этого необходимо добавить краткое его название, краткое описание, указать ответственного и ревьюера (рисунок А.16). После этого выбрать кнопку создания и новый мердж-реквест добавится в список (рисунок А.17).

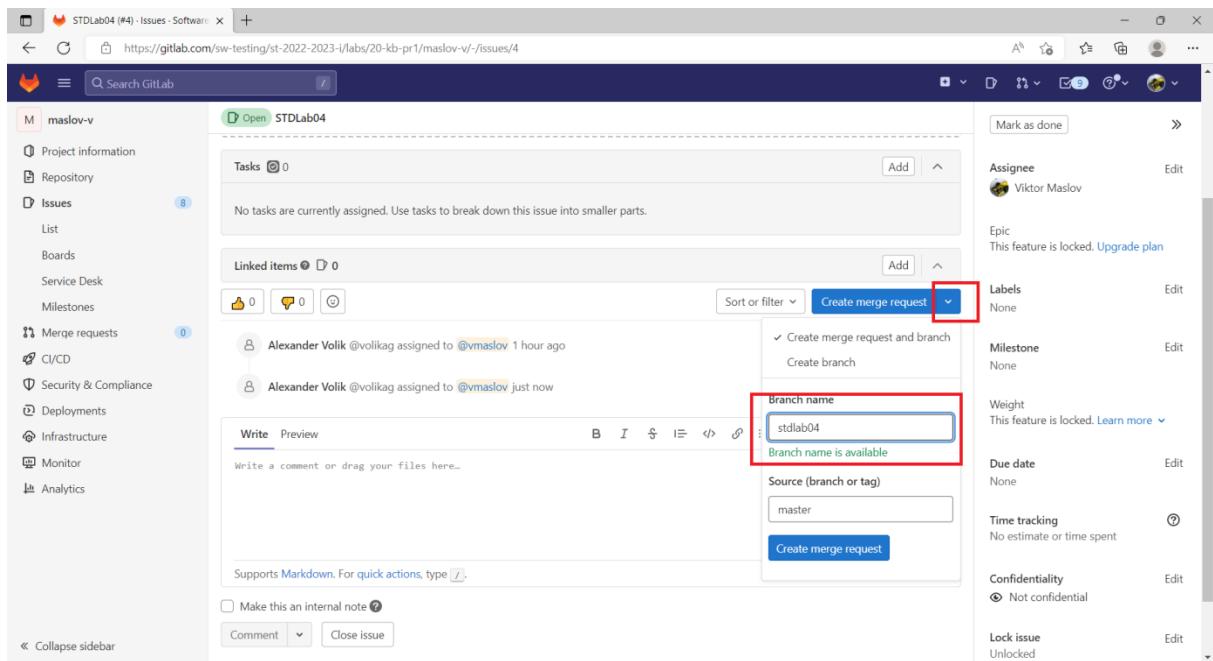


Рисунок А.15 – Создание мердж-реквеста через задачу

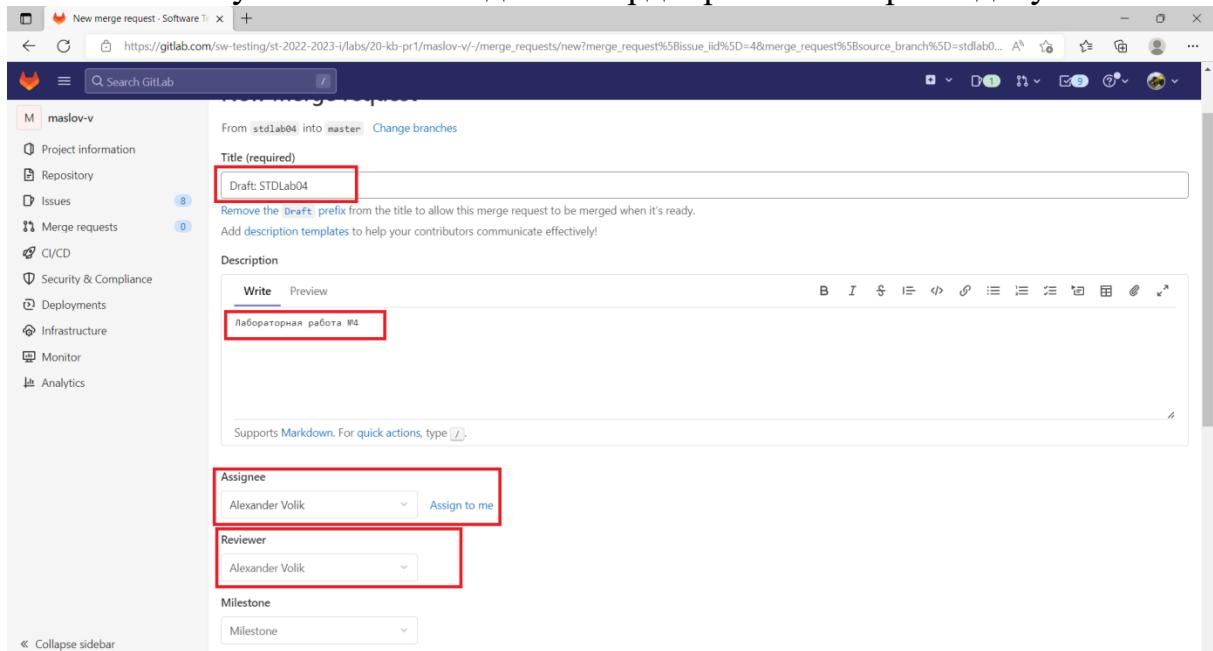


Рисунок А.16 – Заполнение основных полей мердж-реквеста

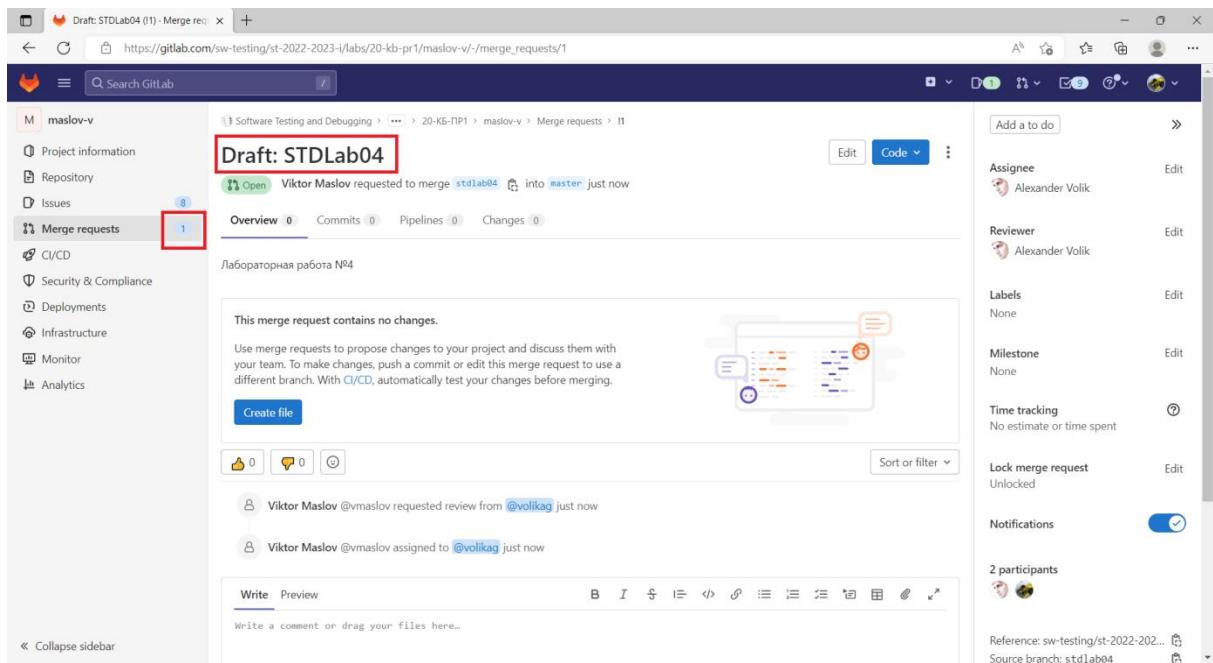


Рисунок А.17 – Результат создания мердж-реквеста

Далее необходимо подтянуть (pull/fetch) изменения в свой локальный репозиторий и переключиться на вновь созданную ветку. Выполнив задания лабораторной работы необходимо залить их обратно в удаленный репозиторий (push). Этой изменения автоматически подтянутся в мердж-реквест (рисунок А.18).

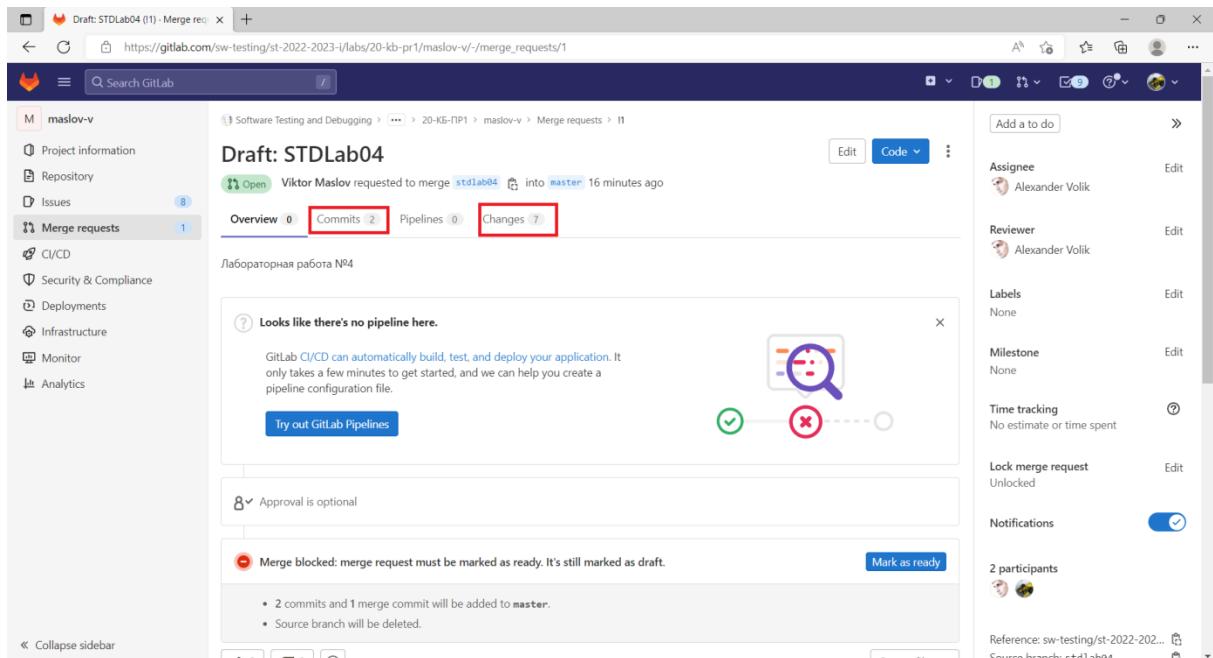


Рисунок А.18 – Изменения в мердж-реквесте

В мердж-реквесте можно посмотреть список коммитов, список изменений и сделать замечания к каждой из строк (рисунок А.19).

The screenshot shows a GitLab interface for a merge request titled 'Draft: STDLab04'. The sidebar on the left lists various project sections like 'Project information', 'Merge requests' (which is selected), 'CI/CD', etc. The main area displays the merge request details: 'Viktor Maslov requested to merge stdlab04 into master 16 minutes ago'. Below this, there are tabs for 'Overview', 'Commits', 'Pipelines', and 'Changes'. The 'Changes' tab is active, showing 7 files with +1001 -0 modifications. A specific file, 'StdLab04/StdLab04/CheckPointInCircle.cs', is expanded, showing its code. The code includes annotations in Russian: '1 + using System;', '2 +', '3 + namespace StdLab04', '4 + {', '5 + public class CheckPointInCircle', '6 + {', '7 + public CheckPointInCircle(double r1, double r2)', '8 + {', '9 + R1 = r1;', '10 + R2 = r2;', '11 + }', '12 +', '13 + public double R1 { get; set; }', '14 +', '15 + public double R2 { get; set; }', '16 +', '17 + //Варианты результатов:', '18 + //0 - точка не принадлежит ни одной из закрашенных областей', '19 + //1 - точка лежит в закрашенную область в круге', '20 + //2 - точка лежит в закрашенную область за окружностью', '21 + public byte CheckOnEntry(double x, double y)'.

Рисунок А.19 – Отображение изменений кода в мердж-реквесте

После того как изменения были залиты и лабораторная готова к проверке, необходимо назначить задачу на преподавателя (как это описано в предыдущем пункте).

В случае возникновения замечаний в мердж-реквесте появляются соответствующие записи (рисунок А.20) указывающие на конкретную строку, вызвавшую вопросы (рисунок А.21).

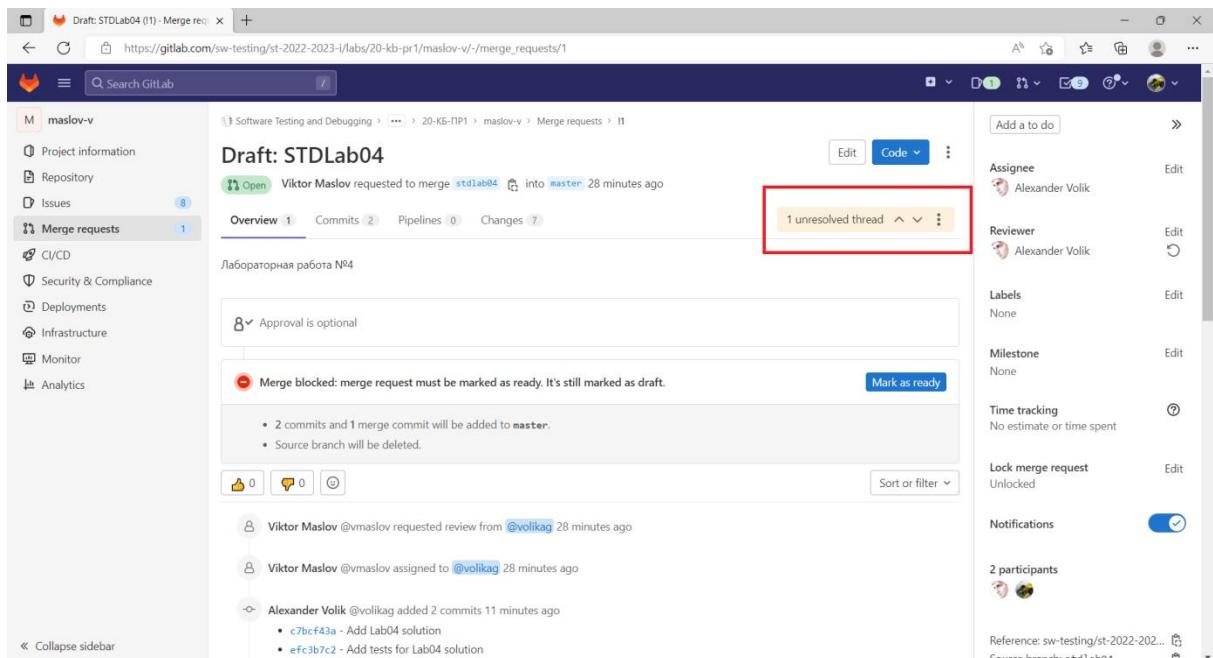


Рисунок А.20 – Отображение замечаний к задаче в мердж-реквесте

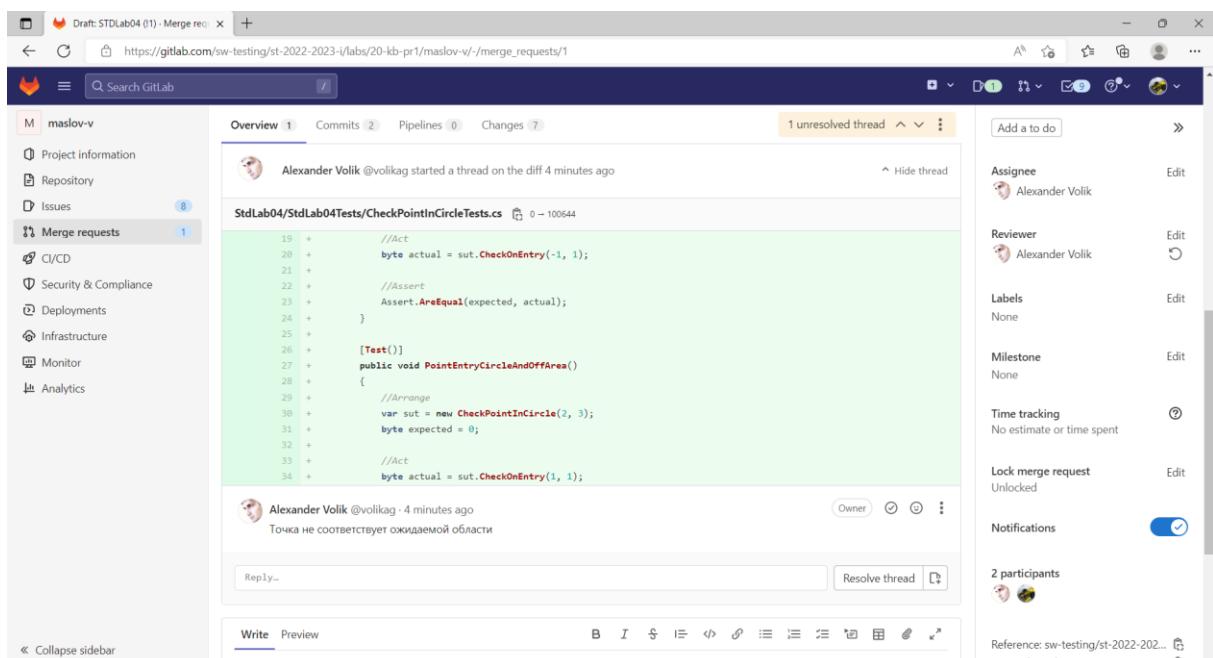


Рисунок А.21 – Отображение фрагмента кода вызвавшего замечания

Получив замечания необходимо устранить их в отдельном коммите, и залить в ту же ветку (рисунок А.22). После этого необходимо оставить комментарий об устранении недочета (рисунок А.23). На данном экране можно заметить фразу *version 2 of the diff*, кликнув на которую можно будет увидеть изменения (рисунок А.24).

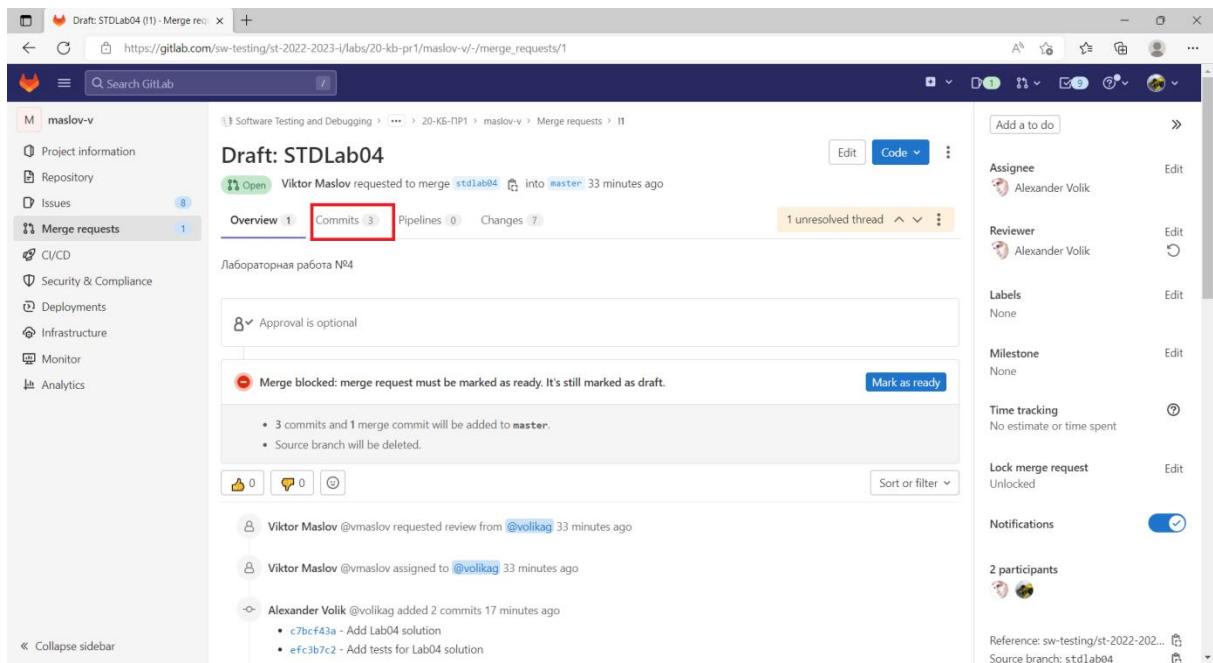


Рисунок А.22 – Заливка исправления в оригиналную ветку

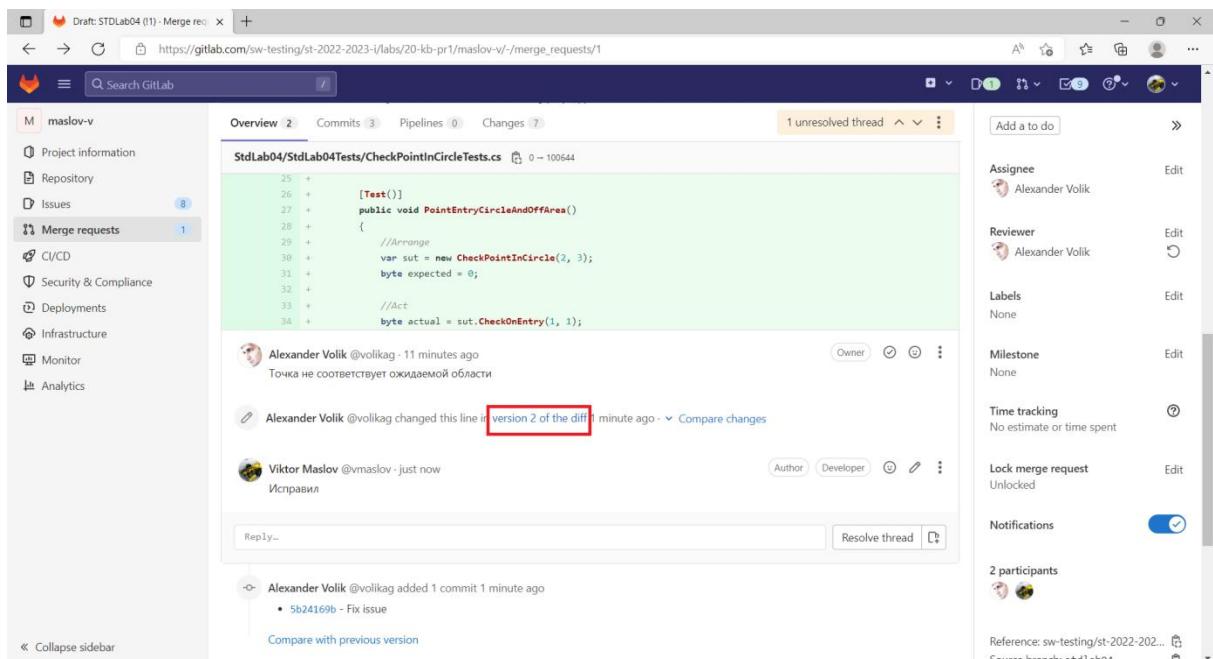


Рисунок А.23 – Добавление комментария об устранении замечания

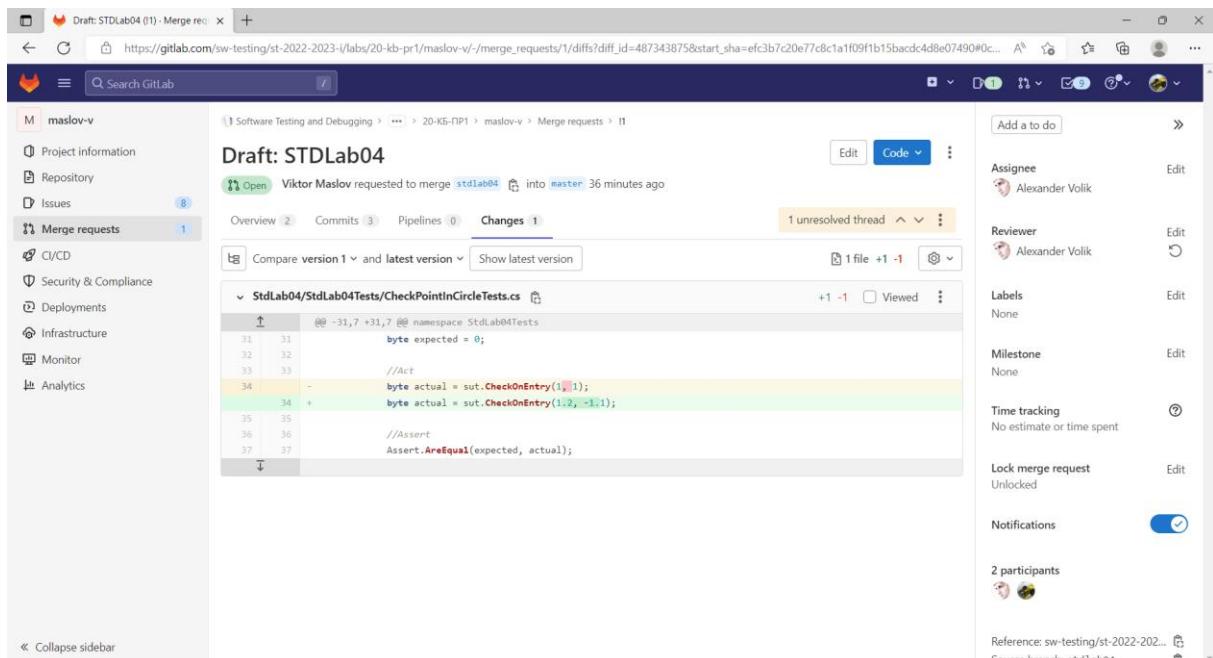


Рисунок А.24 – Просмотр изменений в коде

После того как все изменения были устраниены, преподаватель подтверждает готовность мердж-реквеста (Approved by) и помечает его как готовый (кнопка Mark as ready) (рисунок А.25). После этого он вливает его в основную ветку (рисунок А.26).

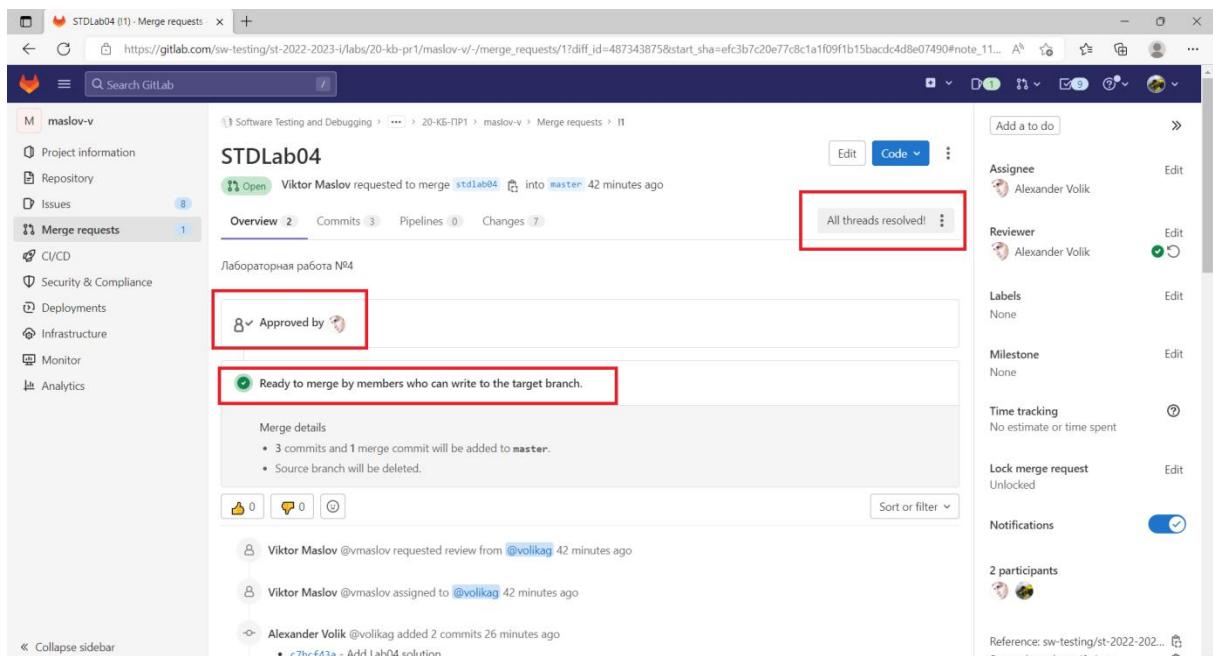


Рисунок А.25 – Подтверждение готовности мердж-реквеста

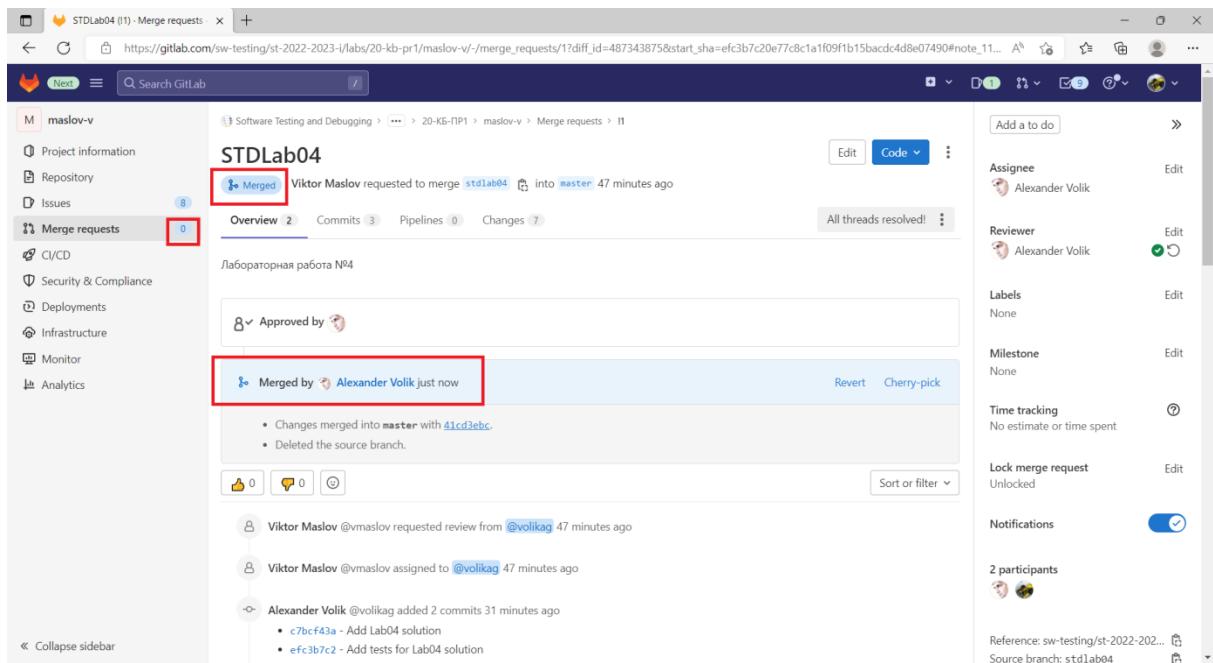


Рисунок А.26 – Влитие мердж-реквеста

4.2 Непосредственное создание Merge Requests

Если ветка была создана ранее без создания мердж-реквеста или вне интерфейса задач на gitlab, то для нее также можно создать отдельный мердж-реквест. Для этого необходимо переключиться в раздел мердж-реквестов и выбрать пункт создания нового мердж-реквеста (рисунок А.27).

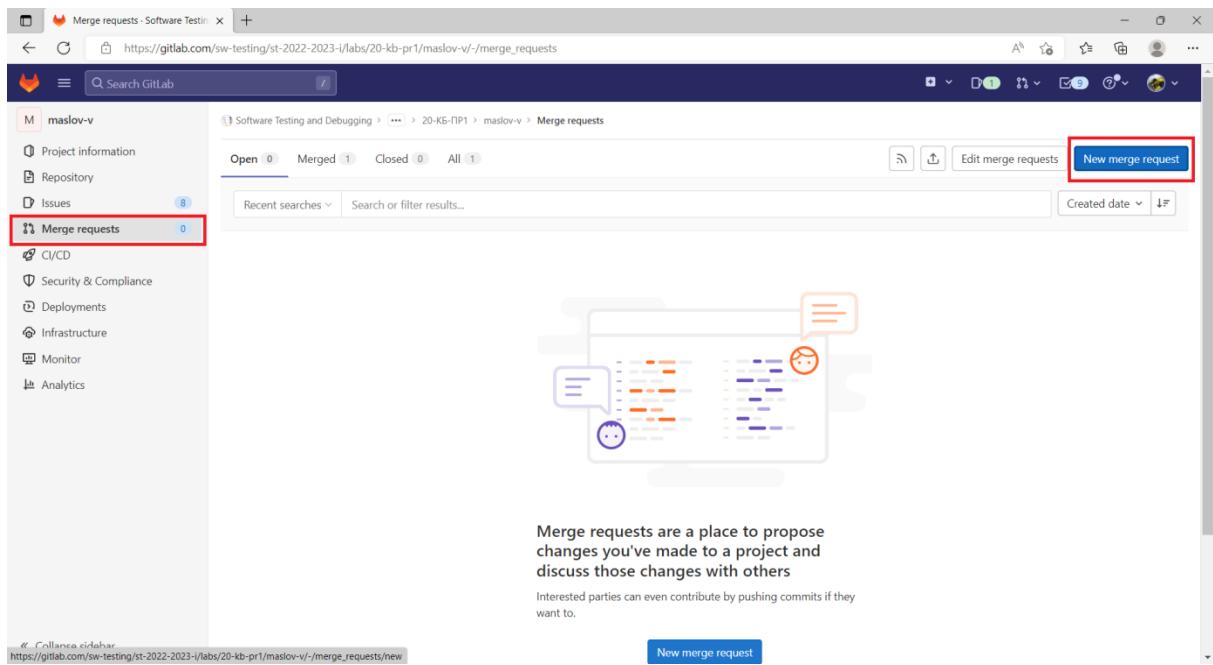


Рисунок А.27 – Создание мердж-реквеста

После этого необходимо выбрать интересующую ветку (Lab05 в нашем случае) и выбрать создания нового мердж-реквеста (рисунок А.28).

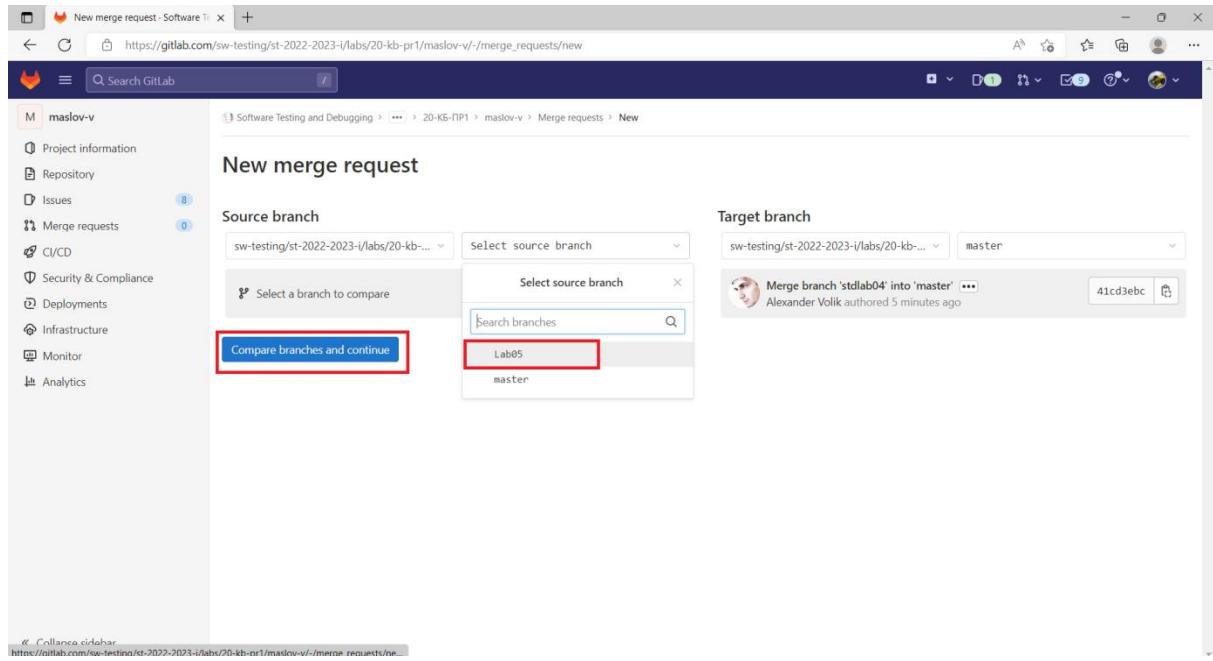


Рисунок А.28 – Выбор ветки для мердж-реквеста

Далее производится заполнение необходимых полей и процесс продолжается как и при создании мердж-реквеста через задачу.

Приложение Б (справочное)

Автоматизация процесса тестирования

Б.1 Автоматизация тестирования

Автоматизация тестирования позволяет получить значительное сокращение времени выполнения повторяющихся тестовых операций (regression testing).

Автоматизированное тестирование программного обеспечения (Software Automation Testing) - это процесс верификации программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования.

Инструмент для автоматизированного тестирования (Automation Test Tool) - это программное обеспечение, посредством которого специалист по автоматизированному тестированию осуществляет создание, отладку, выполнение и анализ результатов прогона тест скриптов.

Тест Скрипт (Test Script) – это набор инструкций, для автоматической проверки определенной части программного обеспечения.

Тестовый набор (Test Suite) – это комбинация тест скриптов, для проверки определенной части программного обеспечения, объединенной общей функциональностью или целями, преследуемыми запуском данного набора.

Тесты для запуска (Test Run) – это комбинация тест скриптов или тестовых наборов для последующего совместного запуска (последовательного или параллельного, в зависимости от преследуемых целей и возможностей инструмента для автоматизированного тестирования).

Существует два основных подхода к автоматизации тестирования: тестирование на уровне кода и GUI-тестирование. К первому типу относится, в частности, модульное тестирование. Ко второму — имитация действий пользователя с помощью специальных тестовых фреймворков.

Одной из главных проблем автоматизированного тестирования является его трудоемкость: несмотря на то, что оно позволяет устраниить часть рутинных операций и ускорить выполнение тестов, большие ресурсы могут тратиться на обновление самих тестов. Это относится к обоим видам автоматизации. При рефакторинге часто бывает необходимо обновить и модульные тесты, и изменение кода тестов может занять столько же времени, сколько и изменение основного кода. С другой стороны, при

изменении интерфейса приложения необходимо заново переписать все тесты, которые связаны с обновленными окнами, что при большом количестве тестов может отнять значительные ресурсы.

Б.2 Модульное тестирование

Модульное тестирование – это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель модульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится в основном по принципу белого или серого ящика, то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования.

Основным принципом создания тестов является Arrange-Act-Assert. Подход Arrange-Act-Assert (AAA) подразумевает, что тест состоит из трех последовательных этапов.

1. Arrange – создать необходимые объекты и данные для тестирования.
2. Act – Вызвать тестируемый метод.
3. Assert – Осуществить проверки.

Таким образом любой разрабатываемый тест должен состоять из этих трех частей.

Б.3 Фреймворк автоматизации модульного тестирования NUnit

NUnit представляет собой свободно распространяемую систему для автоматизации модульного тестирования, применяемую для тестирования программ на платформе .NET.

Модульные тесты, создаваемые с использованием этого фреймворка, представляют собой программные тесты, написанные на языках с поддержкой управляемого кода: C#, Visual Basic, F# или C++.

Модульные тесты используются для выполнения другого исходного кода путем прямого вызова методов класса и передачи соответствующих параметров. Затем, если добавить в код операторы Assert, можно сравнить созданные значения с ожидаемыми. Методы модульных тестов находятся в классах тестов, которые хранятся в файлах исходного кода.

Модульные тесты можно создавать как с помощью средств автоматизации создания кода, создающей первоначальный исходный код теста, так и создавать тесты полностью вручную. В обоих случаях тестовый класс и тестовые методы определяются с помощью программных атрибутов.

Каждый тестовый класс помечен атрибутом [TestFixture()]. Каждый модульный тест – это тестовый метод, помеченный атрибутом [Test()]. Эти атрибуты присваиваются при автоматизации формирования модульных тестов; при создании модульных тестов вручную нужно самостоятельно добавлять атрибуты классов и методов.

Обычно код и атрибуты метода помещают в отдельном файле исходного кода в специально созданном тестовом проекте на том же языке, что и исходный код проекта. Создаваемые тесты предназначены для тестирования определенного фрагмента кода, при этом можно создать одиночный модульный тест для одного метода, несколько тестов для выбранных методов или несколько тестов для всех методов в классе или пространстве имен.

В основном модульные тесты создаются на основе исходного кода текущего проекта. Однако можно создавать модульные тесты для скомпилированной сборки, что полезно, когда исходный код недоступен.

Обычно при создании модульных тестов используется ряд соглашений об именовании. Например, имя тестового файла получается путем соединения слова "Test" с именем файла, содержащего тестируемый код, например Class1Test.cs. Имена тестовых классов и методов также формируются с использованием значений по умолчанию.

Во время создания модульного теста ссылка на пространство имен NUnit.Framework добавляется к тестовому проекту, и пространство имен включается в использующую его инструкцию в верхней части файла модульного теста. Пространство имен содержит много классов для модульных тестов, включая:

- Классы Assert, которые можно использовать, чтобы проверить условия в модульных тестах

- Атрибуты инициализации и очистки для исполнения кода до или после модульных тестов, чтобы обеспечить определенное начальное и конечное состояние

- Атрибут ExpectedException для проверки того, что определенный тип исключения возникает во время исполнения модульного теста

- Класс TestContext, который хранит данные для модульных тестов, например, подключение данных для тестов, управляемых данными, и сведения, необходимые для согласования данных модульных тестов

Во время создания модульного теста TestFixtureAttribute включается в тестовый файл, чтобы показать, что данный конкретный класс может включать методы, помеченные атрибутом [Test()]. Без атрибута TestFixtureAttribute методы теста пропускаются.

Тестовый класс может наследовать методы от другого тестового класса, входящего в ту же сборку. Это означает, что можно создавать методы теста в базовом тестовом классе и затем использовать их в производных тестовых классах.

Во время создания модульных тестов переменная testContextInstance включается для каждого тестового класса. В свойствах класса TestContext хранятся сведения о текущем teste.

Помимо файла модульного теста, создаются и другие файлы.

Какие именно файлы создаются по умолчанию во время создания модульного теста, зависит от параметров тестового проекта.

В фреймворке NUnit существует две модели проверки выполняющегося кода: классическая (classic) и основанная на ограничениях (constraints). В лабораторных мы будем рассматривать классическую модель, основанную на работе класса Assert

Б.4 Класс Assert

Класс Assert пространства имен NUnit.Framework служит для проверки определенных функциональных возможностей. Метод модульного теста использует код метода из кода разработки, но выдает результаты по поведению кода только в том случае, если включены операторы Assert.

Класс Assert содержит следующий набор методов и проверяет условия, использующие утверждения "истина/ложь", в процессе выполнения модульных тестов.

Этот класс содержит набор статических методов, выполняющих оценку логических условий. Если условие оценивается как true, то утверждение проходит проверку.

Assert проверяет предположение истинности для сравниваемых условий и является важной частью процесса модульного теста. Класс

Assert предоставляет множество перегруженных статических методов для проверки предположений истинности утверждений (assertions). Если проверяемое условие является ложным, то утверждение не выполняется.

В таблице Б.1 приведены основные группы перегруженных методов класса Assert. Более детально параметры перегруженных методов могут быть получены из документации к фреймворку NUnit.

Кроме этого существует целые группы дополнительных утверждений для работы со строками, коллекциями, файлами и каталогами.

Таблица Б.1 – Основные группы перегруженных методов класса Assert

Общее имя группы	Описание
Проверка равенства (Equality Asserts)	
AreEqual	Проверяет два объекта на равность. Утверждение выполняется, если объекты равны.
AreNotEqual	Проверяет два объекта на неравность. Утверждение выполняется, если объекты не равны.
Проверка сущностей (Identity Asserts)	
AreNotSame	Проверяет, ссылаются ли две указанные объектные переменные на разные объекты. Утверждение не выполняется, если переменные ссылаются на один и тот же объект.
AreSame	Проверяет, ссылаются ли две указанные объектные переменные на один и тот же объект. Утверждение не выполняется, если переменные ссылаются на разные объекты.
Contains	Проверяет, содержится ли указанный объект в коллекции IList. Утверждение выполняется, если объект не содержится в коллекции.
Проверка сравнений (Comparison Asserts)	
Greater	Проверяет что первый аргумент больше второго
GreaterOrEqual	Проверяет что первый аргумент больше или равен второму
Less	Проверяет что первый аргумент меньше второго
LessOrEqual	Проверяет что первый аргумент меньше или равен второму
Проверка типов (Type Asserts)	
IsInstanceOfType	Проверяет, является ли данный объект экземпляром некоторого типа. Утверждение выполняется, если объекты является экземпляром
IsNotInstanceOfType	Проверяет, является ли данный объект экземпляром некоторого типа. Утверждение выполняется, если объекты не является экземпляром
IsAssignableFrom	Проверяет, возможность приведения объекта к указателю некоторого типа. Утверждение выполняется, если присвоение возможно
IsNotAssignableFrom	Проверяет, возможность приведения объекта к указателю некоторого типа. Утверждение выполняется, если присвоение не

	возможно
--	----------

Продолжение таблицы Б.1

Условные проверки (Condition tests)	
IsTrue	Проверяет, имеет ли указанное условие значение true. Утверждение не выполняется, если условие имеет значение false.
IsFalse	Проверяет, имеет ли указанное условие значение false. Утверждение не выполняется, если условие имеет значение true.
True	Проверяет, имеет ли указанное условие значение true. Утверждение не выполняется, если условие имеет значение false.
False	Проверяет, имеет ли указанное условие значение false. Утверждение не выполняется, если условие имеет значение true.
IsNull	Проверяет, имеет ли указанный объект значение null. Утверждение не выполняется, если объект имеет значение отличное от null.
Null	Проверяет, имеет ли указанный объект значение null. Утверждение не выполняется, если объект имеет значение отличное от null.
IsNotNull	Проверяет, не имеет ли указанный объект значение null. Утверждение не выполняется, если объект имеет значение null.
NotNull	Проверяет, не имеет ли указанный объект значение null. Утверждение не выполняется, если объект имеет значение null.
IsNaN	Проверяет, является ли указанное число с плавающей точкой не числом (NaN – Not a Number)
IsEmpty	Проверяет, является ли указанная строка или коллекция пустой
IsNotEmpty	Проверяет, является ли указанная строка или коллекция не пустой
Утилиты (Utility methods)	
Pass	Подтверждает выполнение утверждения без проверки каких-либо условий.
Fail	Отменяет выполнение утверждения без проверки каких-либо условий.
Ignore	Указывает, что утверждение будет проигнорировано.
Inconclusive	Указывает, что утверждение не может быть проверено.
Проверка исключений (Exception Asserts)	
Throws	Проверяет, генерируется ли заданное исключение в ходе работы тестируемого метода
DoesNotThrow	Проверяет, что исключение в ходе работы тестируемого метода не генерируется
Catch	Работает как Throws за исключение того, что проверяет также производные классы исключений

Б.5 Основные результаты теста

При выполнении теста создаются результаты, которые отображаются в окне результатов теста. Некоторые результаты теста являются общими для всех типов тестов. Другие результаты формируются только по определенным типам теста или имеют особое значение в зависимости от типа теста, из которого они образуются. В таблице Б.2 описаны возможные результаты выполнения тестов в среде Visual Studio.

Таблица Б.2 – Результаты выполнения тестов

Результат	Ситуация возникновения
Прерван (Aborted)	Тестер остановил тестовый запуск. Выполняемому тесту назначается состояние "Прерван". Остальным тестам в тестовом запуске присваивается состояние "Не выполнен"
Не выполнен (Not Executed)	Тестер остановил тестовый запуск. Выполняемому тесту назначается состояние "Прерван". Остальным тестам в тестовом запуске присваивается состояние "Не выполнен"
Пройден (выполнение прервано) (Passed But Run Aborted)	Выполнен и пройден отдельный тест. По завершении выполнения этого теста тестер остановил тестовый запуск
Неработоспособен (Not Runnable)	Не удалось выполнить тест из-за ошибок в определении теста. Например, модульный тест может быть неработоспособным, если он возвращает целое число; методы модульных тестов должны возвращать пустое значение
Отключен (Disconnected)	Это удаленный запуск, который сначала отключается, а затем подключается. Этот результат возникает при отключении удаленного запуска. Когда тестер подключается к этому удаленному запуску, он может видеть результаты тестов.
Истекло время ожидания (Timeout)	Истекло время ожидания теста или тестового запуска
Отложен (Pending)	Тестовый запуск был начат и выполняется, но выполнение отдельного теста не было завершено
Выполняется (In Progress)	Тест выполняется
Завершен (Completed)	Тестовый запуск завершен. Этот результат применяется только к нагрузочным тестам
С неопределенным результатом (Inconclusive)	При запуске теста оператор Assert не выдал результат "Ошибка", и, по меньшей мере, один оператор Assert.Inconclusive был выполнен успешно. Этот результат применим только к модульным тестам
Ошибка (Failed)	При запуске теста, по меньшей мере, один оператор Assert выдал результат "Ошибка" или тест выдал непредвиденное исключение

Пройден (Passed)	При запуске теста ни один оператор Assert не выдал неопределенный результат или ошибку, и тест не выдал непредвиденного исключения, и время ожидания не истекло
---------------------	---

Б.6 Подготовка к созданию тестового проекта

Перед тем как приступить к созданию тестов с помощью фреймворка NUnit необходимо выполнить ряд подготовительных действий. В дальнейшем, на уже настроенном компьютере, повторять этот этап заново не потребуется.

Для начала необходимо установить сам фреймворк. Скачать его можно с официального сайта проекта <http://nunit.org> или с GitHub <https://github.com/nunit/nunit>. Актуальной версией фреймворка является версия 3.12.

Примечание. По желанию студенты могут выбрать другие тестовые фреймворки, например xUnit или MS Test. Работа с ними в целом мало отличается от NUnit, поэтому не должна вызывать больших проблем.

Рекомендуемым Microsoft способом установки всех сторонних библиотек является использование nuget-пакетов, работающих только в рамках конкретного решения (solution).

Более подробно описание работы с nuget приведено в приложении А.

В среде разработки JetBrains Rider имеется встроенная поддержка NUnit.

Для поддержки запуска тестов из среды разработки Visual Studio, кроме самого тестового фреймворка, необходимо установить nugget-пакет с официальным расширением от разработчиков NUnit: «NUnit Test Adapter».

Помимо nuget-пакета для Visual Studio 2017 и 2019 «NUnit Test Adapter» также можно установить глобально (для всех проектов Visual Studio) через менеджер расширений (Extensions and Updates) или скачать с сайта Visual Studio Gallery:

<http://visualstudiogallery.msdn.microsoft.com/6ab922d0-21c0-4f06-ab5f-4ecd1fe7175d/>

Б.7 Пример создания проекта для модульного тестирования

Б.7.1 Создание тестового проекта в JetBrains Rider 2018

Рассмотрим проект, содержащий в своем составе класс, реализующий методы поиска минимального и максимального элемента в массиве.

Создадим отдельный проект с основной программой (рисунок Б.1 – Б.3) и добавим класс, выполняющий основную функциональность (рисунок Б.4 и рисунок Б.5).

Примечание. Приведенные ниже экраны могут немного отличаться, однако описанные действия в целом будут совпадать для различных версий IDE.

При этом все же необходимо следить за версиями используемых библиотек и фреймворков, а так же группами решений (solutions). В приведенном случае необходимо выбрать консольное приложение (Console Application) из группы решений «.NET» и версию фреймворка «.Net Framework 4.7.2».

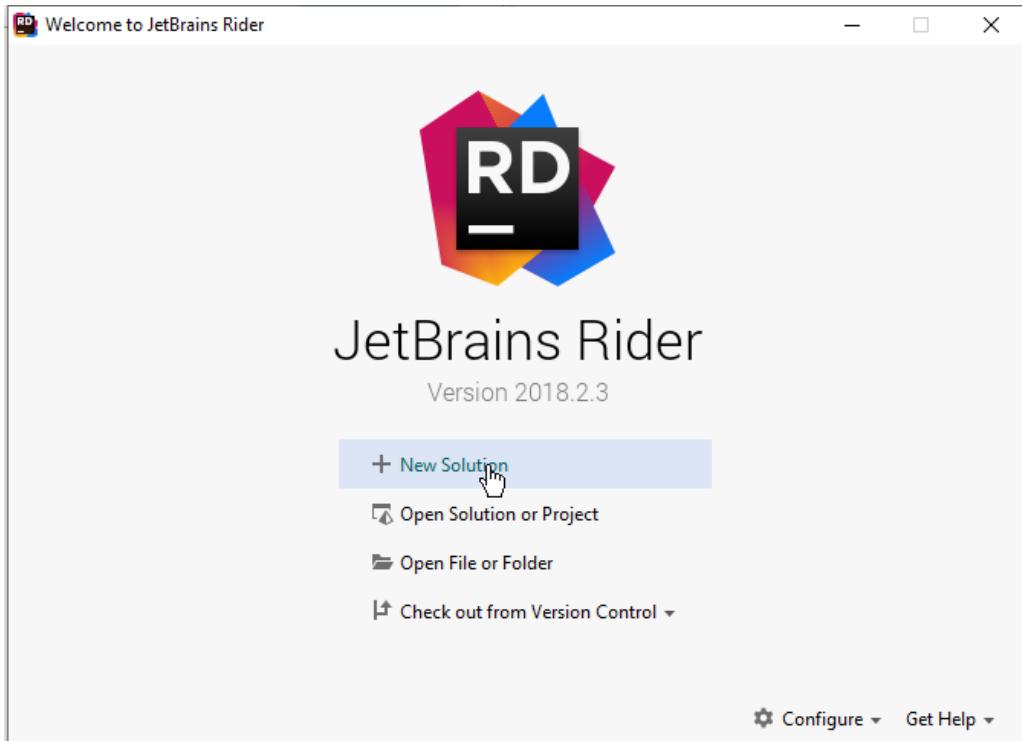


Рисунок Б.1 – Стартовый экран JetBrains Rider

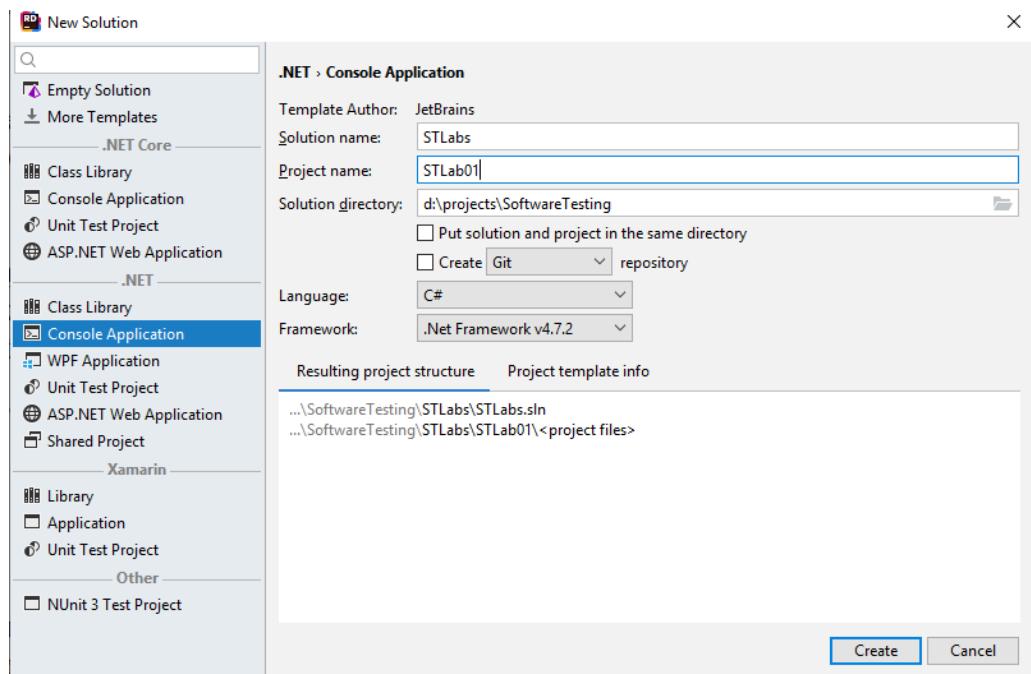


Рисунок Б.2 – Выбор типа и настроек проекта

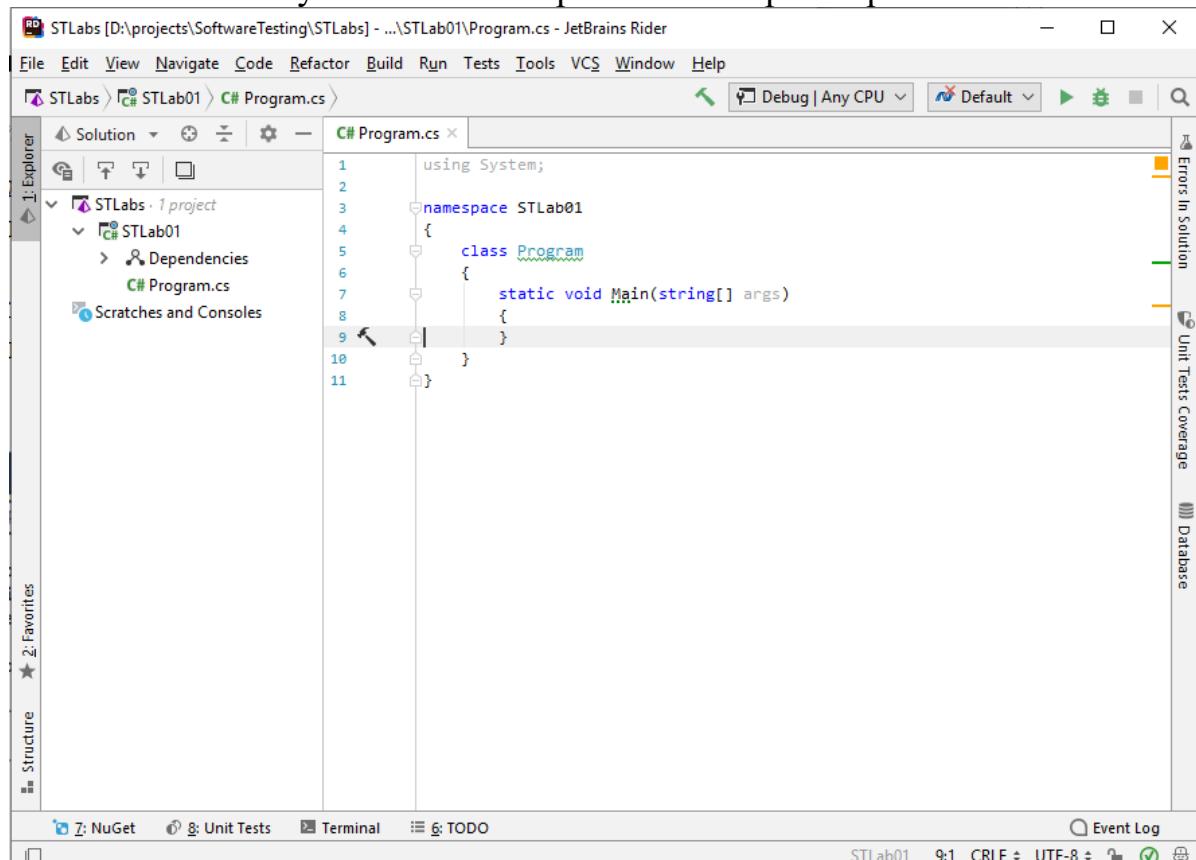


Рисунок Б.3 – Результат создания проекта основной программы

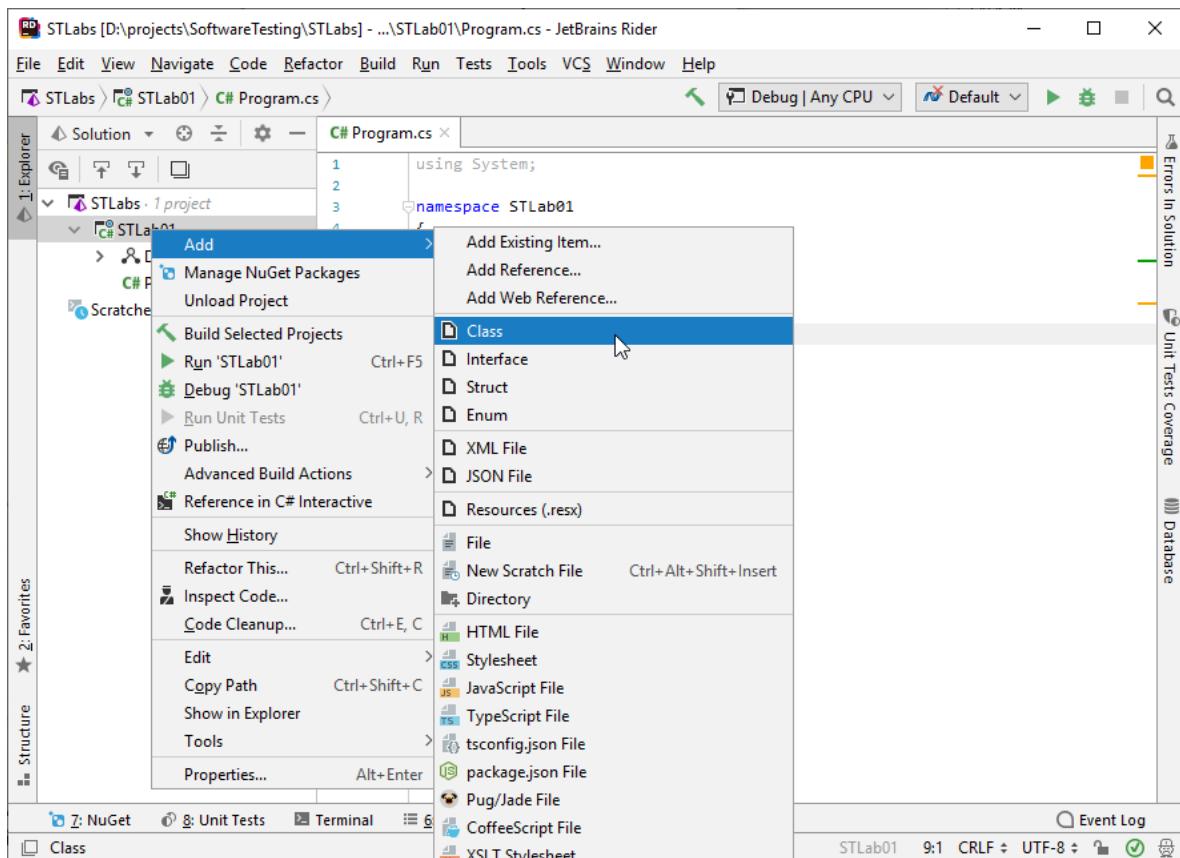


Рисунок Б.4 – Добавление класса

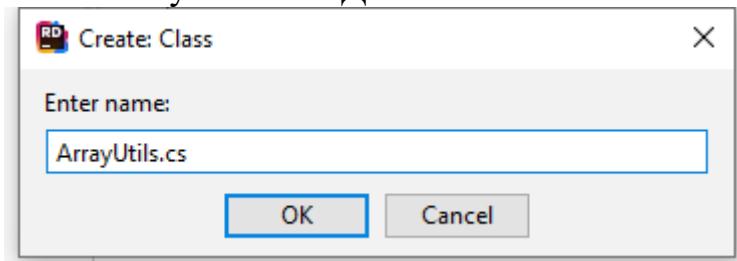


Рисунок Б.5 – Добавление нового класса

Далее реализуем основные методы класса, листинг которого представлен ниже.

Пример Б.1. Класс для модульного тестирования

```
using System;

namespace STLab01
{
    public class ArrayUtils
    {
        public static int Min(int[] a)
        {
            if (a == null)
                throw new

```

```

        ArgumentException("Ошибка. В метод передан пустой массив.");

        int min = a[0];
        foreach (int x in a)
        {
            if (min > x)
                min = x;
        }
        return min;
    }

    public static int Max(int[] a)
    {
        if (a == null)
            throw new
        ArgumentException("Ошибка. В метод передан пустой массив.");
        int max = a[0];
        foreach (int x in a)
        {
            if (max < x)
                max = x;
        }
        return max;
    }
}

```

Существует несколько способов добавить поддержку тестирования в свой проект.

Один из них заключается в добавлении нового тестового проекта в решение (рисунок Б.6). В этом случае создается новый пустой проект библиотеки классов без учета содержимого текущего проекта. Далее необходимо ввести имя нового тестового проекта. Имя выбирается в соответствии со следующими соглашениями: к имени проекта добавляется слово «Test» (рисунок Б.7).

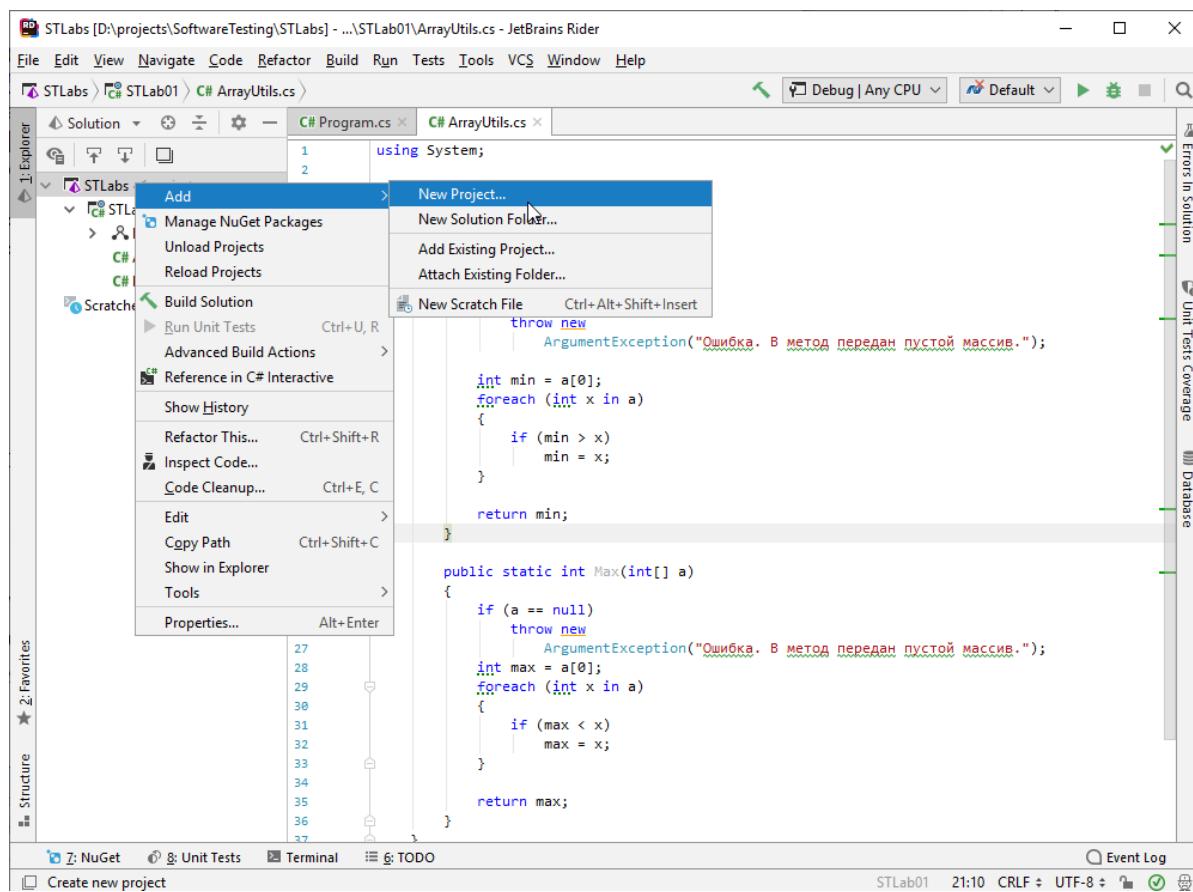


Рисунок Б.6 – Добавление нового тестового проекта

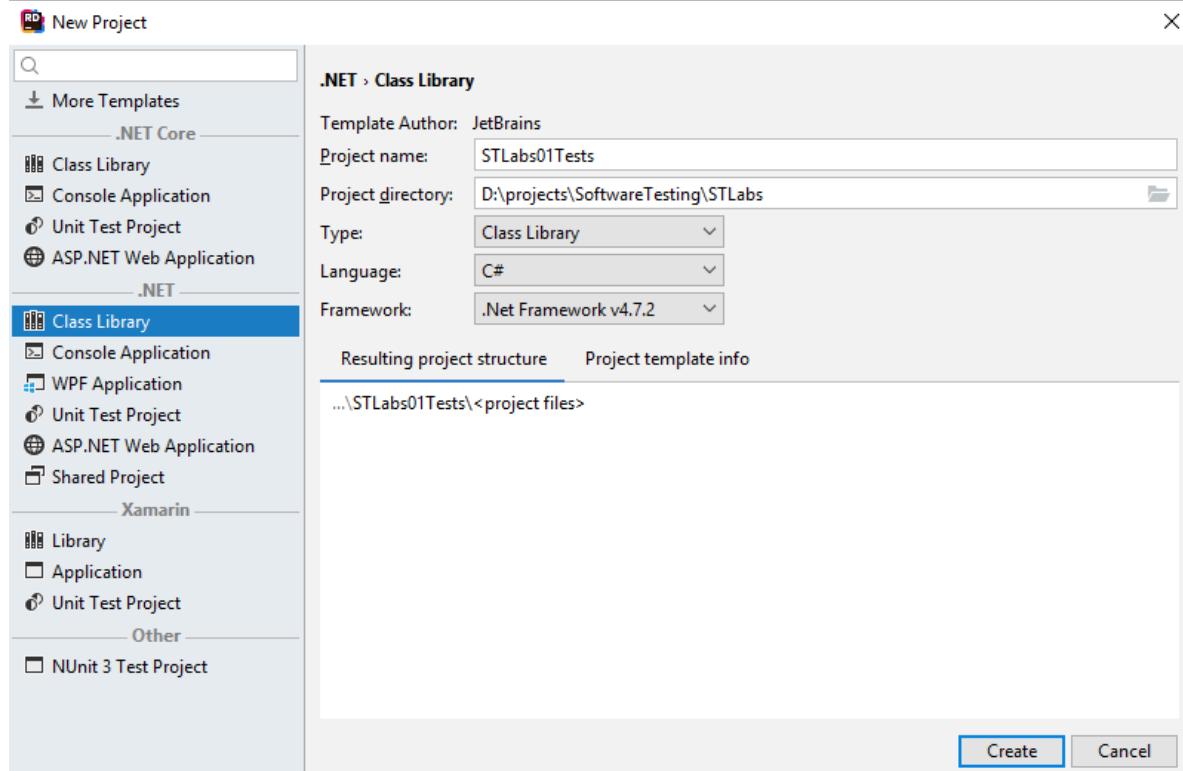


Рисунок Б.7 – Настройки нового тестового проекта

После этого необходимо поменять имя файла тестов и содержащегося в нем класса, в соответствии с принятыми соглашениями: к имени тестируемого класса необходимо добавить слово Test. При этом для каждого класса, подвергаемого тестированию, в тестовом проекте создается файл модульного теста.

На рисунке ниже показан обозреватель решений после создания модульного теста для нашего примера проекта (рисунок Б.8).

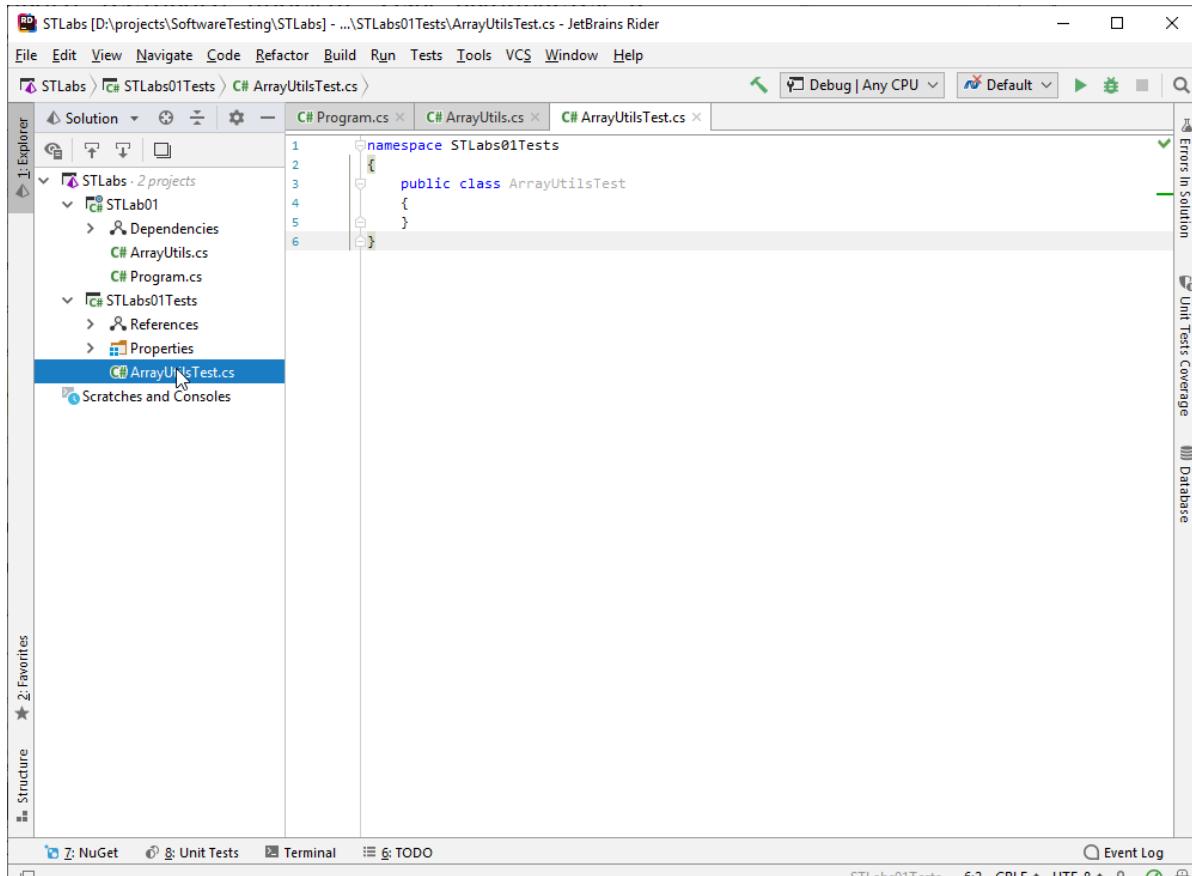


Рисунок Б.8 – Обозреватель решений для примера проекта модульного теста

Тестовый проект содержит файлы, необходимые для модульных тестов. Свойства тестового проекта содержат файл AssemblyInfo.cs, в котором содержатся параметры построения проекта.

Далее необходимо настроить ссылки на сборки NUnit для получения возможности использования фреймворка. Для этого необходимо подключить соответствующую сборку из nuget-пакета (рисунки Б.9-Б.13).

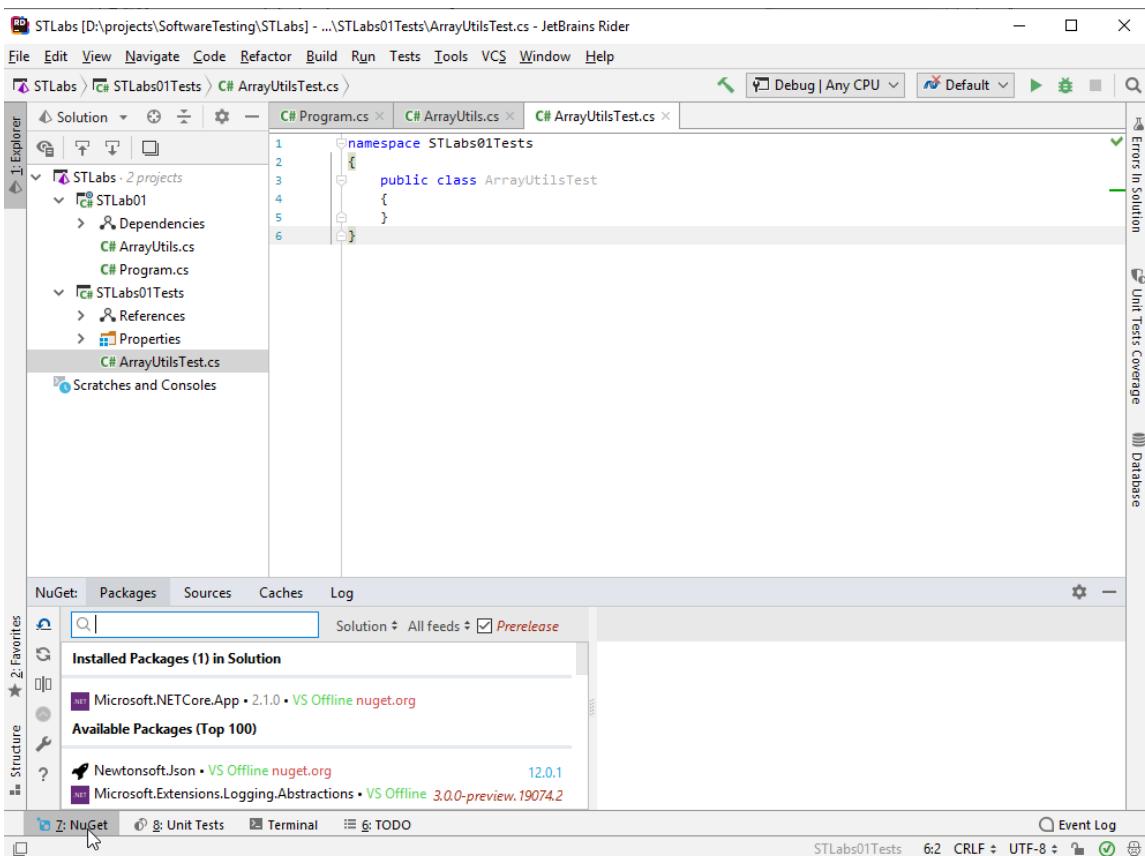


Рисунок Б.9 – Открытие вкладки управления nuget-пакетами

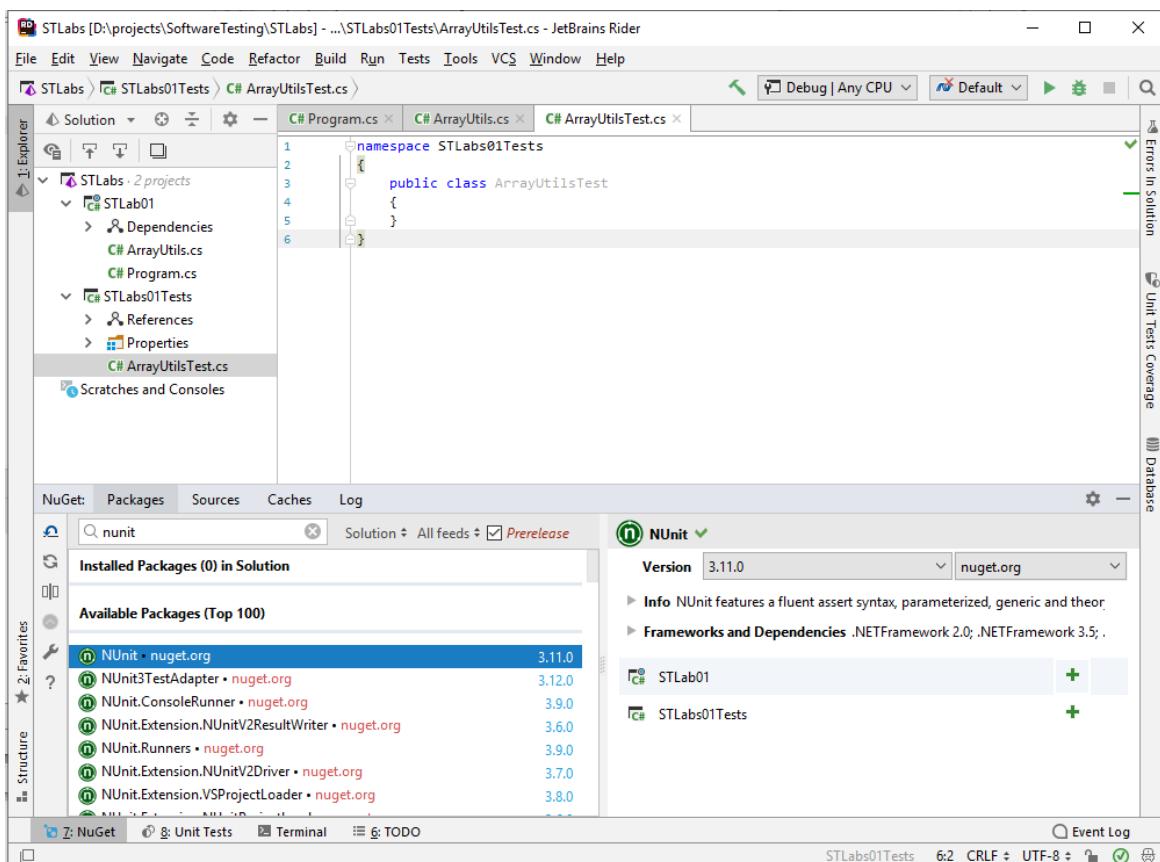


Рисунок Б.10 – Поиск nuget-пакета

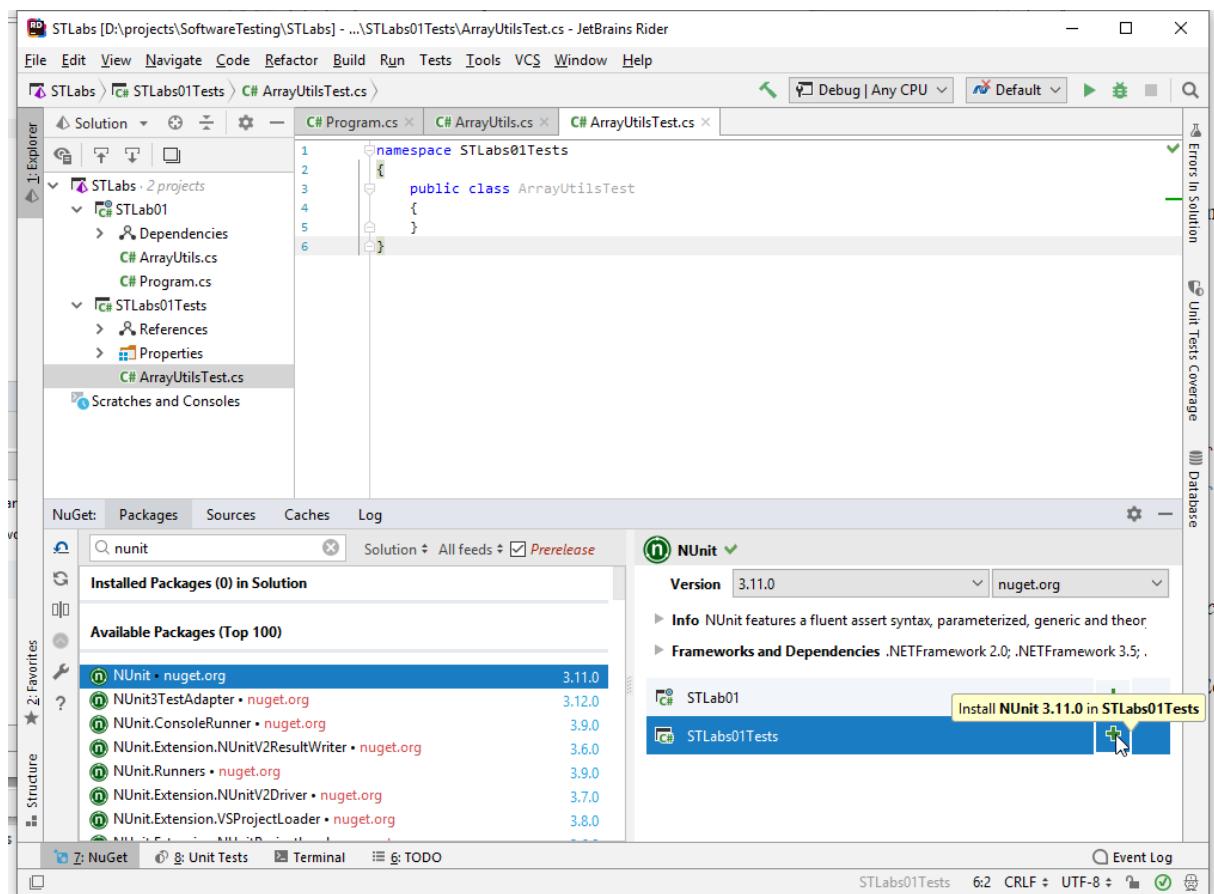


Рисунок Б.11 – Добавление nuget-пакета в проект

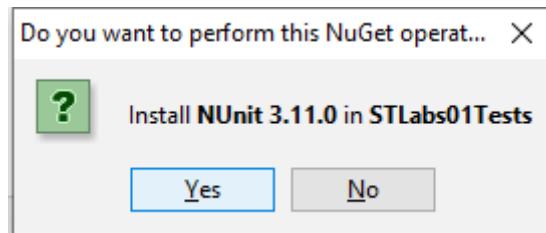


Рисунок Б.12 – Подтверждение добавления nuget-пакета в проект

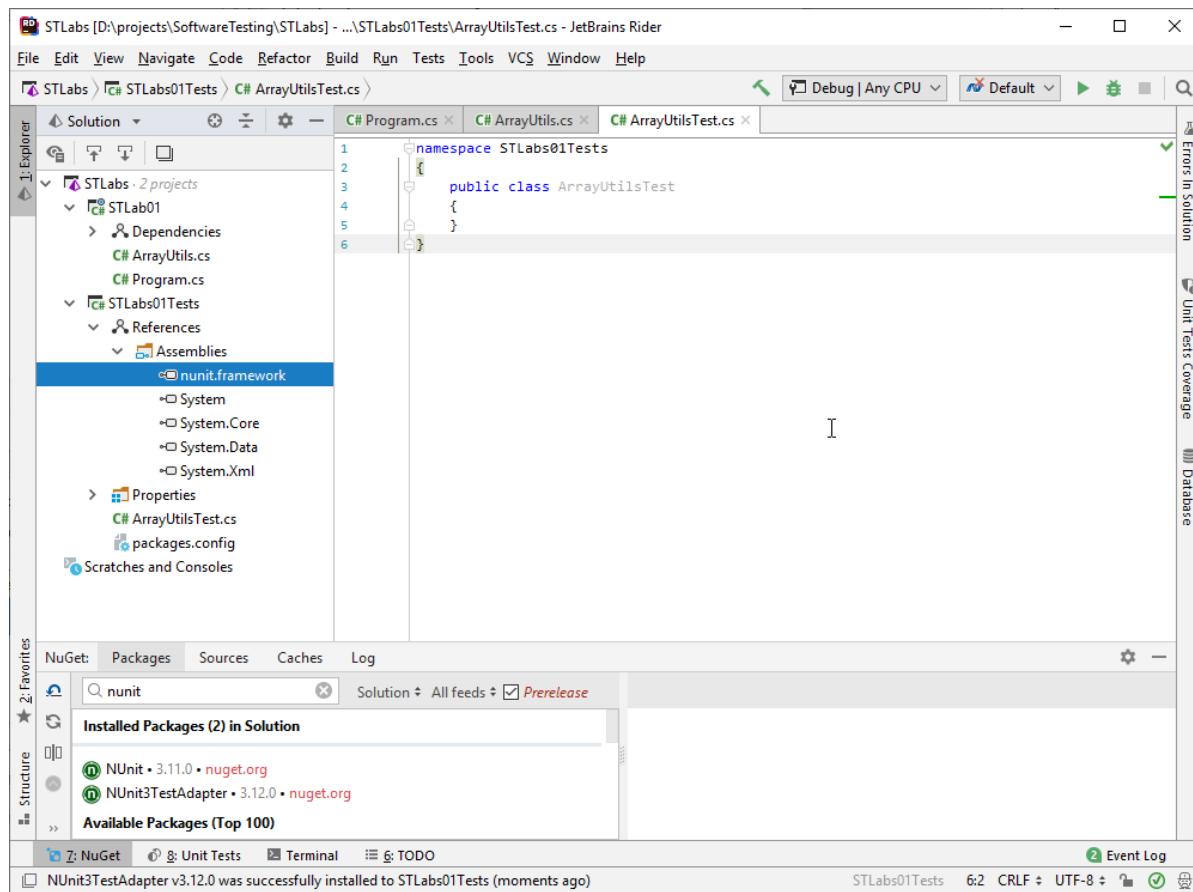


Рисунок Б.13 – Результат добавления nuget-пакета NUnit в проект

Кроме этого необходимо добавить ссылку на сборку тестируемого проекта (рисунки Б.14-Б.16).

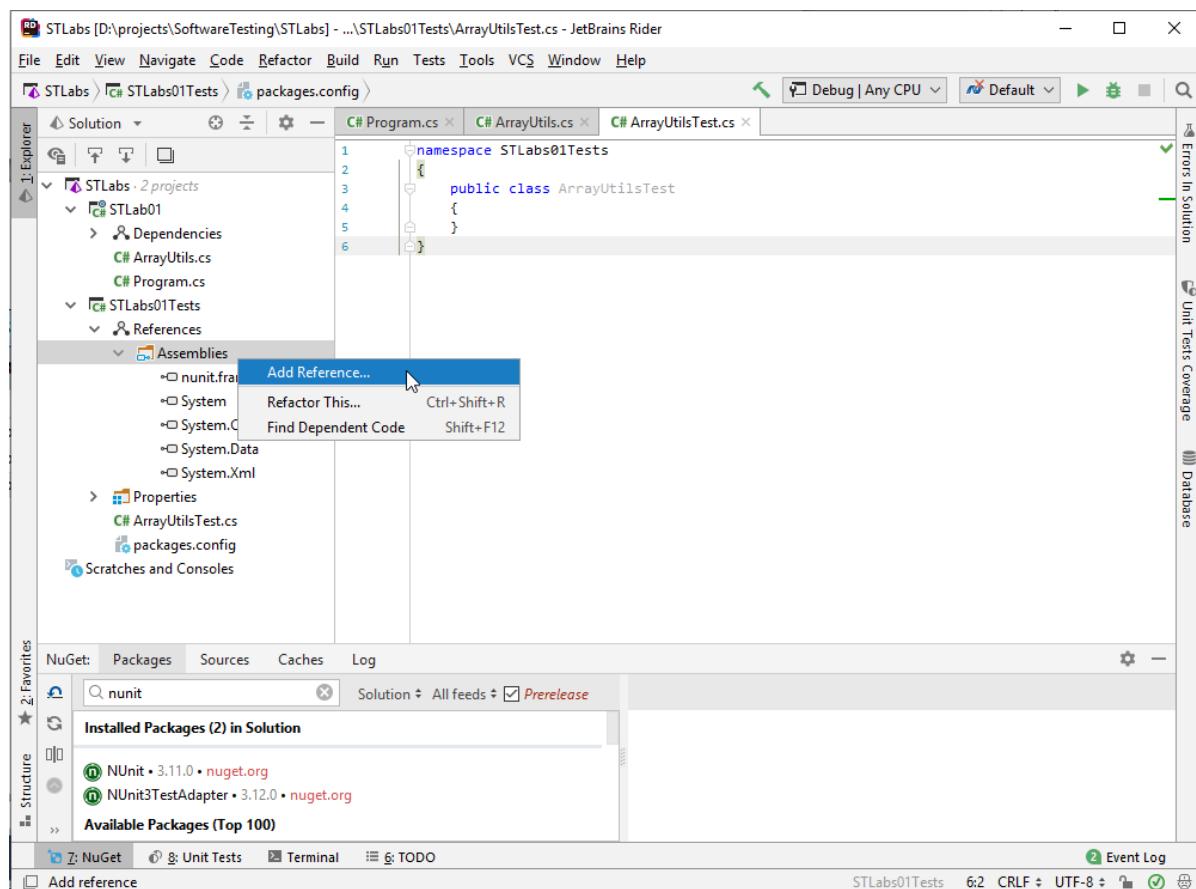


Рисунок Б.14 – Открытие окна добавления ссылки на проект

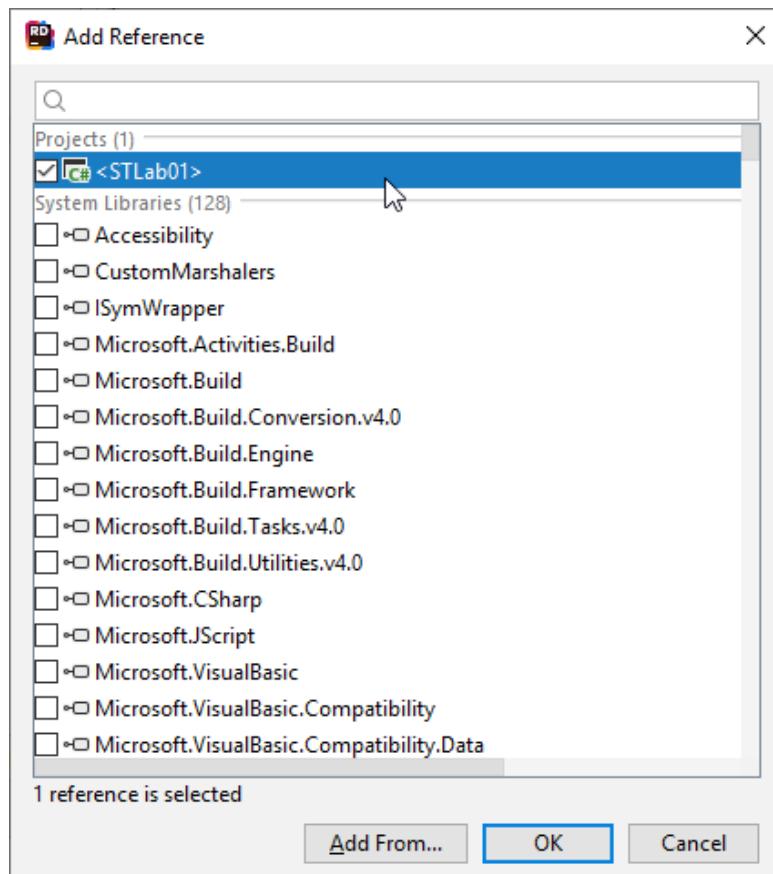


Рисунок Б.15 – Добавление сборки основного проекта

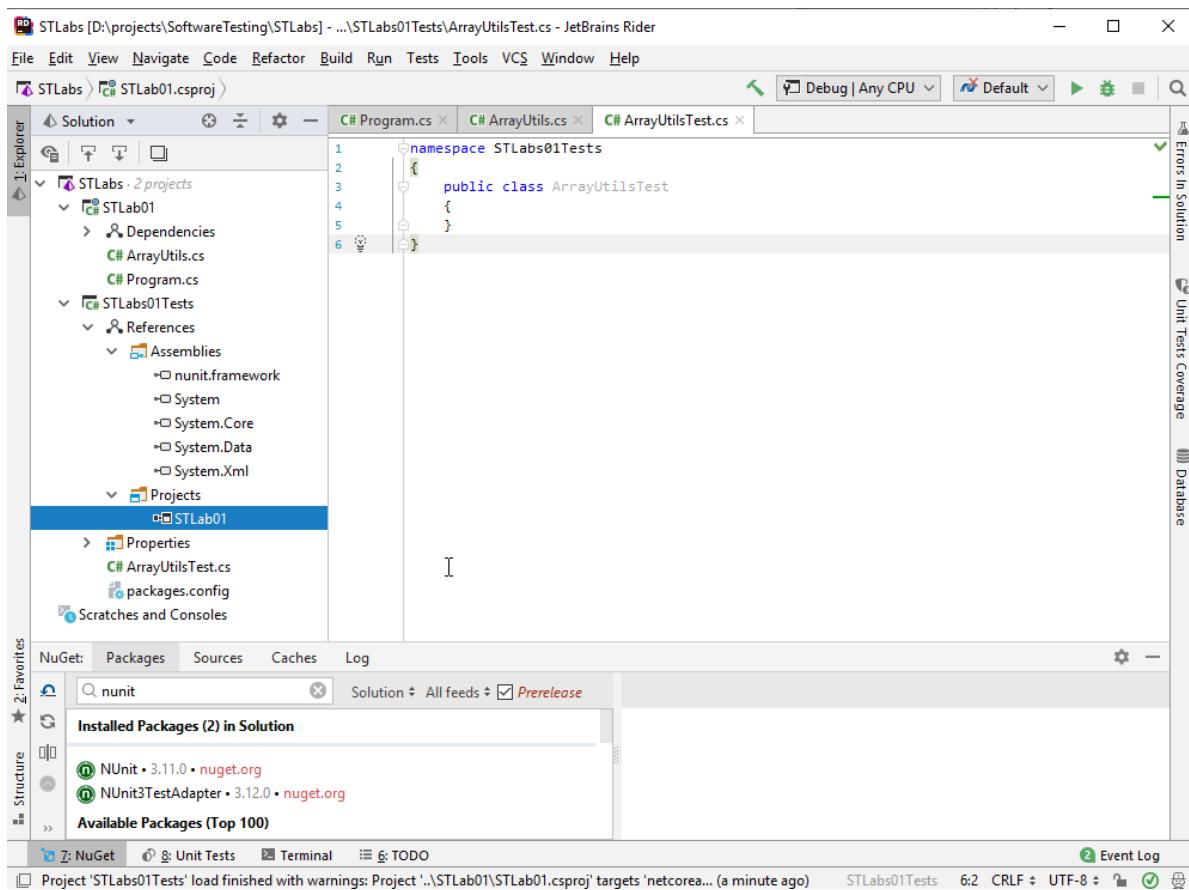


Рисунок Б.16 – Результат добавления сборки тестируемого проекта

Далее в коде тестового класса необходимо при помощи директивы `using` подключить пространство имен `NUnit.Framework` и пространство имен тестируемого проекта.

Исходя из требований, можно выделить следующие классы эквивалентности входных значений:

- 1) Массив из нескольких различных элементов (правильный вариант)
- 2) Массив из нескольких одинаковых элементов (правильный вариант)
- 3) Массив из нескольких различных элементов содержащих повторения (правильный вариант)
- 4) Массив из одного элемента (правильный вариант)
- 5) Пустой массив (ошибочный вариант)

На основе этих классов можно составить по пять тестов для каждого варианта. Каждый тест представляет собой отдельный тестовый метод добавляемый отдельно. При этом имена тестов могут быть любыми неповторяющимися идентификаторами языка. Обычно при выборе имени пользуются следующим правилом: вначале идет имя тестируемого метода, за ним слово `Test` и далее описание теста в формате «что проверяется» и

«ожидаемый результат». Примеры именования приведены в коде для тестирования метода Min (пример Б.2).

Создаваемый тестовый класс должен быть помечен атрибутом [TestFixture()], а все тестовые методы атрибутом [Test()].

Напишем тесты с учетом того, что для обработки ошибочного варианта потребуется обработка соответствующего исключения, генерируемого основным кодом тестируемой программы. Для этого в тестовом методе необходимо использовать Assert.Throws с указанием класса исключения:

```
Assert.Throws<ArgumentException>(
    () => { actual = ArrayUtils.Min(a); }
);
```

Следует обратить внимание на выражение «() => { ... }».

Оно представляет собой анонимную функцию (лямбда-функцию), которая может объявляться в месте своего вызова.

Т.е. данное выражение эквивалентно созданию обычной функции и передачи ее в качестве параметра для Assert.Throws.

Далее добавим код, необходимый для получения минимального набора тестов.

Пример Б.2. Незавершённый модульный тест

```
using System;

using NUnit.Framework;
using STLab01;

namespace STLab01Test
{
    [TestFixture]
    public class ArrayUtilsTest
    {
        [Test()]
        public void MinTestNormalArrayWithUniqueElements()
        {
            //Класс эквивалентности 1
            int[] a = { 1, 2, 9, -4, 5 };
            int expected = -4;
            int actual;
            actual = ArrayUtils.Min(a);
            Assert.AreEqual(expected, actual);
        }

        [Test()]
        public void MinTestNormalArrayWithSameElements()
```

```

{
    //Класс эквивалентности 2
    int[] a = { 2, 2, 2 };
    int expected = 2;
    int actual;
    actual = ArrayUtils.Min(a);
    Assert.AreEqual(expected, actual);
}

[TestMethod]
public void MinTestNormalArrayWithRepeatedElements()
{
    //Класс эквивалентности 3
    int[] a = { 1, 2, 2, 2, 1, 4 };
    int expected = 1;
    int actual;
    actual = ArrayUtils.Min(a);
    Assert.AreEqual(expected, actual);
}

[TestMethod]
public void MinTestArrayWithOneElement()
{
    //Класс эквивалентности 4
    int[] a = { 3 };
    int expected = 3;
    int actual;
    actual = ArrayUtils.Min(a);
    Assert.AreEqual(expected, actual);
}

[TestMethod]
public void MinTestEmptyArray()
{
    //Класс эквивалентности 5 (ошибочный)
    int[] a = null;
    int actual;
    Assert.Throws<ArgumentException>(
        () => { actual = ArrayUtils.Min(a); })
}

[TestMethod]
public void MaxTest()
{
    int[] a = null;
    int expected = 0;
    int actual;
    actual = ArrayUtils.Max(a);
    Assert.AreEqual(expected, actual);
}

```

```
    }  
}
```

Для того чтобы тесты были добавлены в план тестирования необходимо открыть вкладку Unit Tests (рисунок Б.17). Оно открывается через пункт меню View => Tool Windows => Unit Tests. Запуск тестов может осуществляться либо через контекстное меню вкладки Unit Tests (рисунок Б.18), либо через пункт меню «Tests».

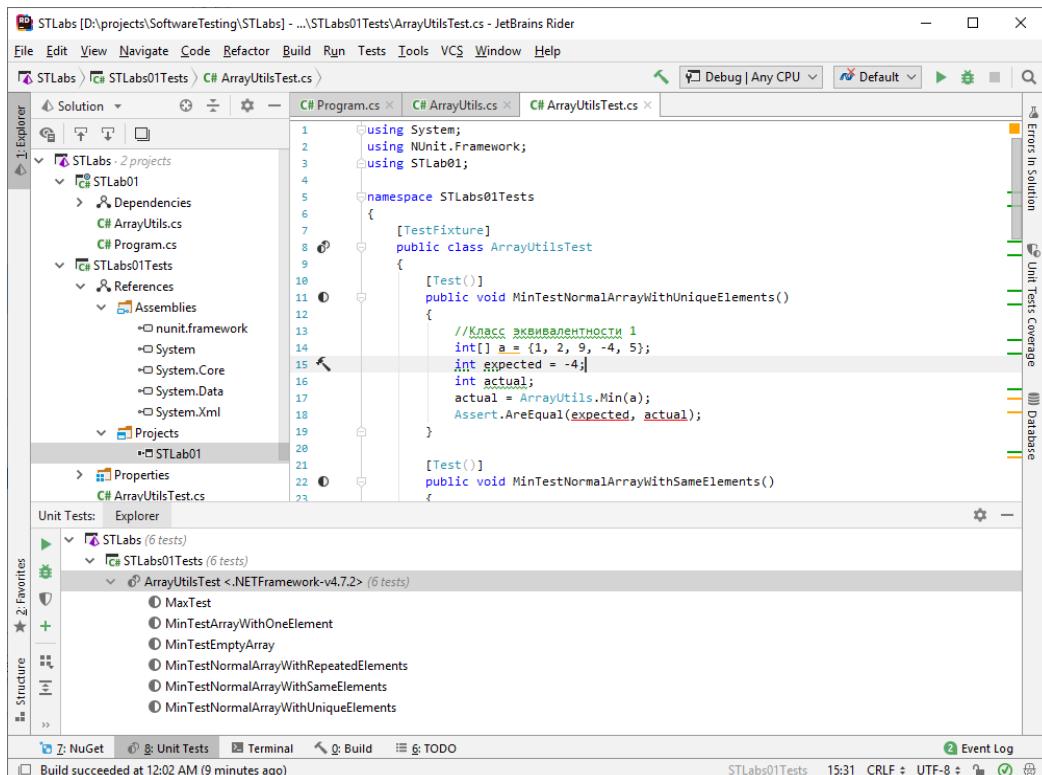


Рисунок Б.17 – Вкладка Unit Tests

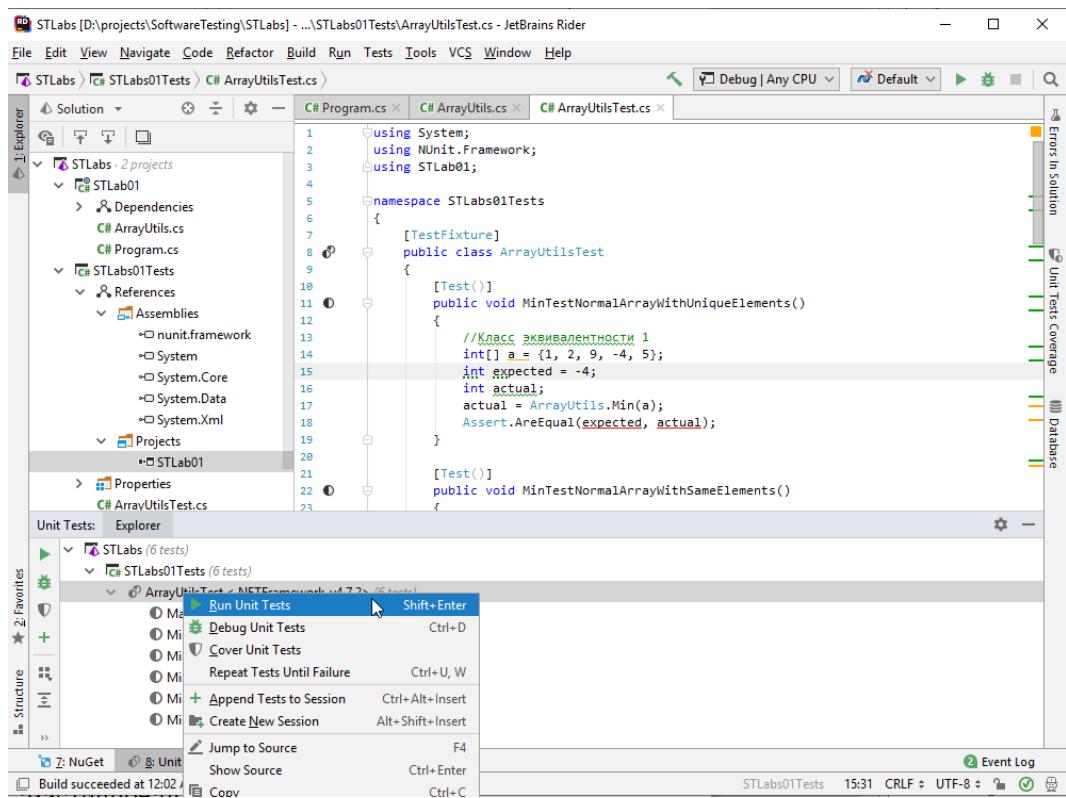


Рисунок Б.18 – Запуск тестов

После этого запустив набор созданных тестов (касающихся метода Min) будет пройдена, а часть останется (для метода Max) (рисунок Б.19).

После успешного прохождения теста он исключается из списка для дальнейших запусков до внесения изменений в код.

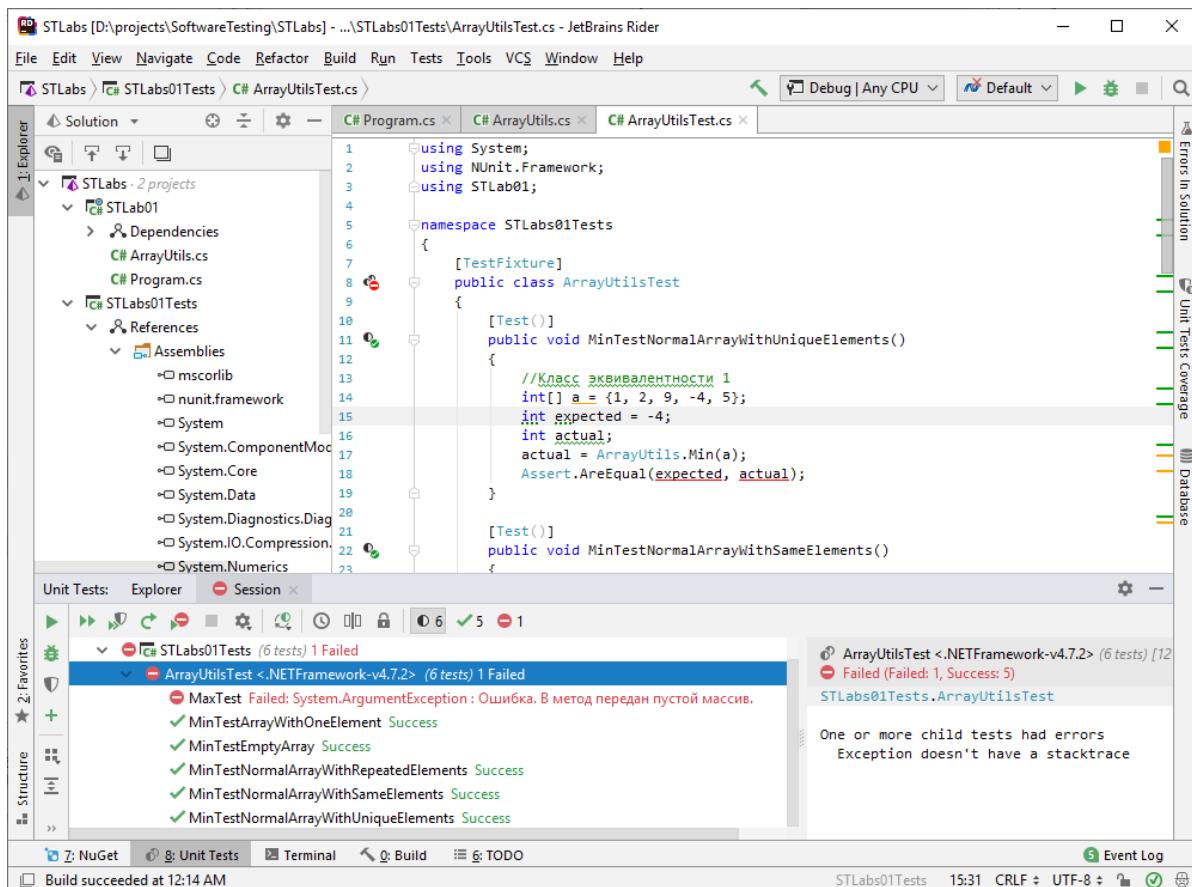


Рисунок Б.19 – Частичное прихождение тестов

Допишем оставшиеся тесты для метода Max. Ниже приведен листинг полного модульного теста.

Пример Б.3 Полный модульный тестовый набор

```

using System;

using NUnit.Framework;
using STLab01;

namespace STLab01Test
{
    [TestFixture]
    public class ArrayUtilsTest
    {
        // Тестирование метода Min
        [Test()]
        public void MinTestNormalArrayWithUniqueElements()
        {
            //Класс эквивалентности 1
            int[] a = { 1, 2, 9, -4, 5 };
            int expected = -4;
            int actual;
            Assert.AreEqual(expected, actual);
        }
    }
}

```

```

        actual = ArrayUtils.Min(a);
        Assert.AreEqual(expected, actual);
    }

    [Test()]
    public void MinTestNormalArrayWithSameElements()
    {
        //Класс эквивалентности 2
        int[] a = { 2, 2, 2 };
        int expected = 2;
        int actual;
        actual = ArrayUtils.Min(a);
        Assert.AreEqual(expected, actual);
    }

    [Test()]
    public void MinTestNormalArrayWithRepeatedElements()
    {
        //Класс эквивалентности 3
        int[] a = { 1, 2, 2, 2, 1, 4 };
        int expected = 1;
        int actual;
        actual = ArrayUtils.Min(a);
        Assert.AreEqual(expected, actual);
    }

    [Test()]
    public void MinTestArrayWithOneElement()
    {
        //Класс эквивалентности 4
        int[] a = { 3 };
        int expected = 3;
        int actual;
        actual = ArrayUtils.Min(a);
        Assert.AreEqual(expected, actual);
    }

    [Test()]
    public void MinTestEmptyArray()
    {
        //Класс эквивалентности 5 (ошибочный)
        int[] a = null;
        int actual;
        Assert.Throws<ArgumentException>(
            () => { actual = ArrayUtils.Min(a); }
        );
    }

    // Тестирование метода Max
    [Test()]

```

```

public void MaxTestWithUniqueElements()
{
    //Класс эквивалентности 1
    int[] a = { 2, 4, 3, 10 };
    int expected = 10;
    int actual;
    actual = ArrayUtils.Max(a);
    Assert.AreEqual(expected, actual);
}

[Test()]
public void MaxTestWithSameElements()
{
    //Класс эквивалентности 2
    int[] a = { 8, 8, 8 };
    int expected = 8;
    int actual;
    actual = ArrayUtils.Max(a);
    Assert.AreEqual(expected, actual);
}

[Test()]
public void MaxTestWithRepeatedElement()
{
    //Класс эквивалентности 3
    int[] a = { 3, 6, 9, 44, 44, 3 };
    int expected = 44;
    int actual;
    actual = ArrayUtils.Max(a);
    Assert.AreEqual(expected, actual);
}

[Test()]
public void MaxTestArrayWithOneElement()
{
    //Класс эквивалентности 4
    int[] a = { 3 };
    int expected = 3;
    int actual;
    actual = ArrayUtils.Max(a);
    Assert.AreEqual(expected, actual);
}

[Test()]
public void MaxTestEmptyArray()
{
    //Класс эквивалентности 5 (ошибочный)
    int[] a = null;
    int actual;
    Assert.Throws<ArgumentException>(

```

```

        () => { actual = ArrayUtils.Max(a); }
    );
}
}

```

В итоге мы получим полностью проходящий тесты класс (рисунок Б.20) и теперь привнесении любых изменений сможем определить, не внесли ли мы какую либо ошибку, просто запустив тесты заново.

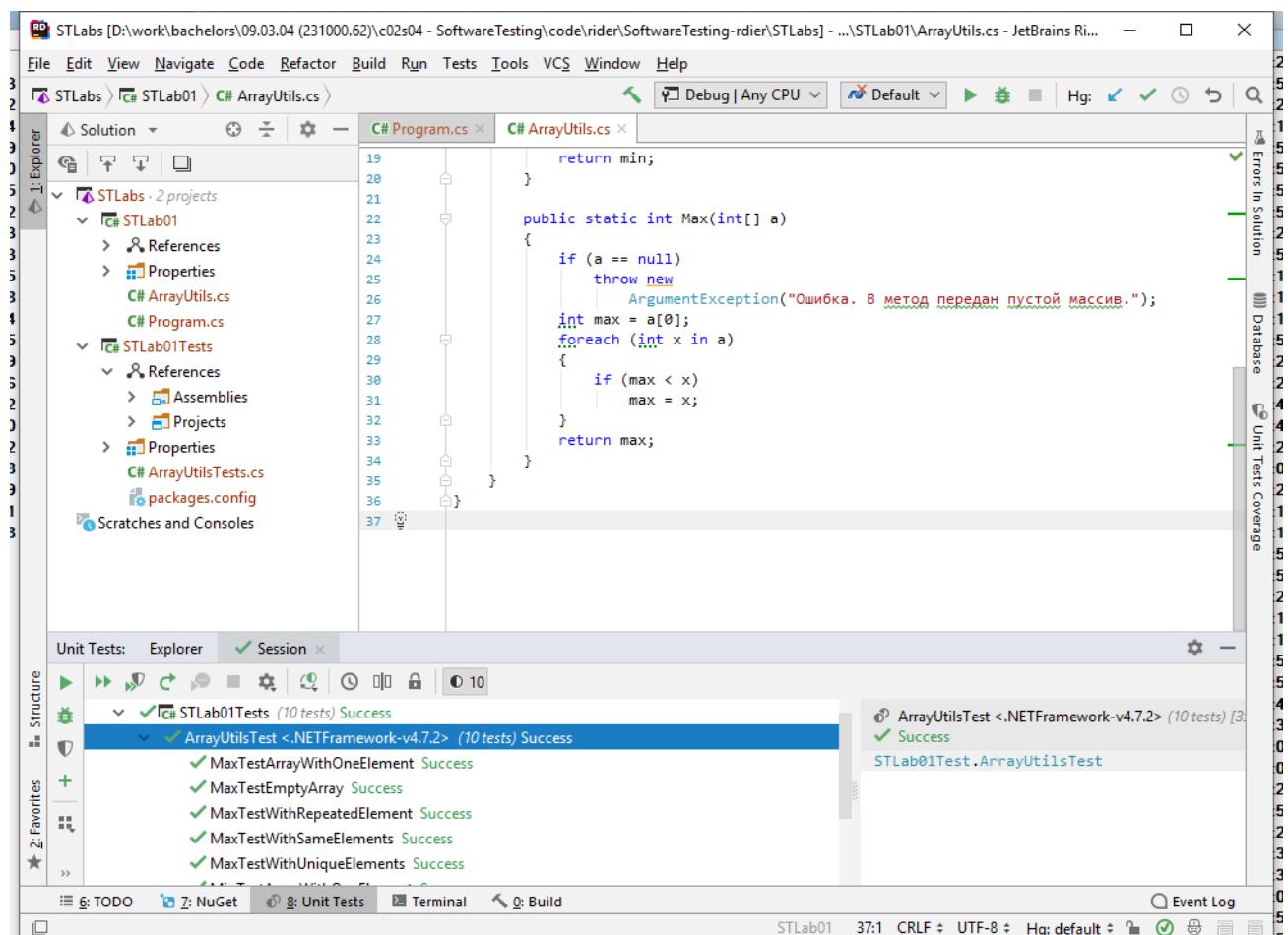


Рисунок Б.20 – Полное прихождение тестов

Б.7.2 Создание тестового проекта в Visual Studio 2017/2019

Создадим в среде Visual Studio аналогичный проект с основной программой (рисунок Б.21) и добавим класс, выполняющий основную функциональность (рисунок Б.22 и рисунок Б.23).

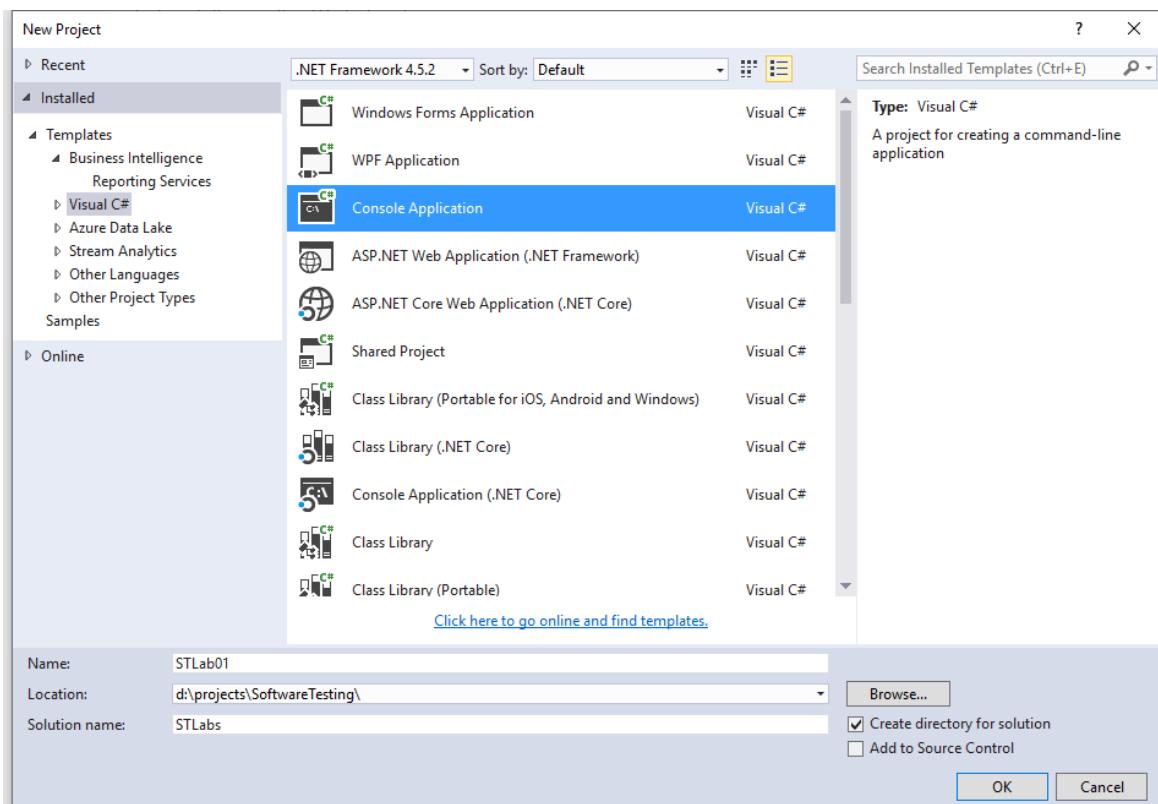


Рисунок Б.21 – Создание проекта основной программы

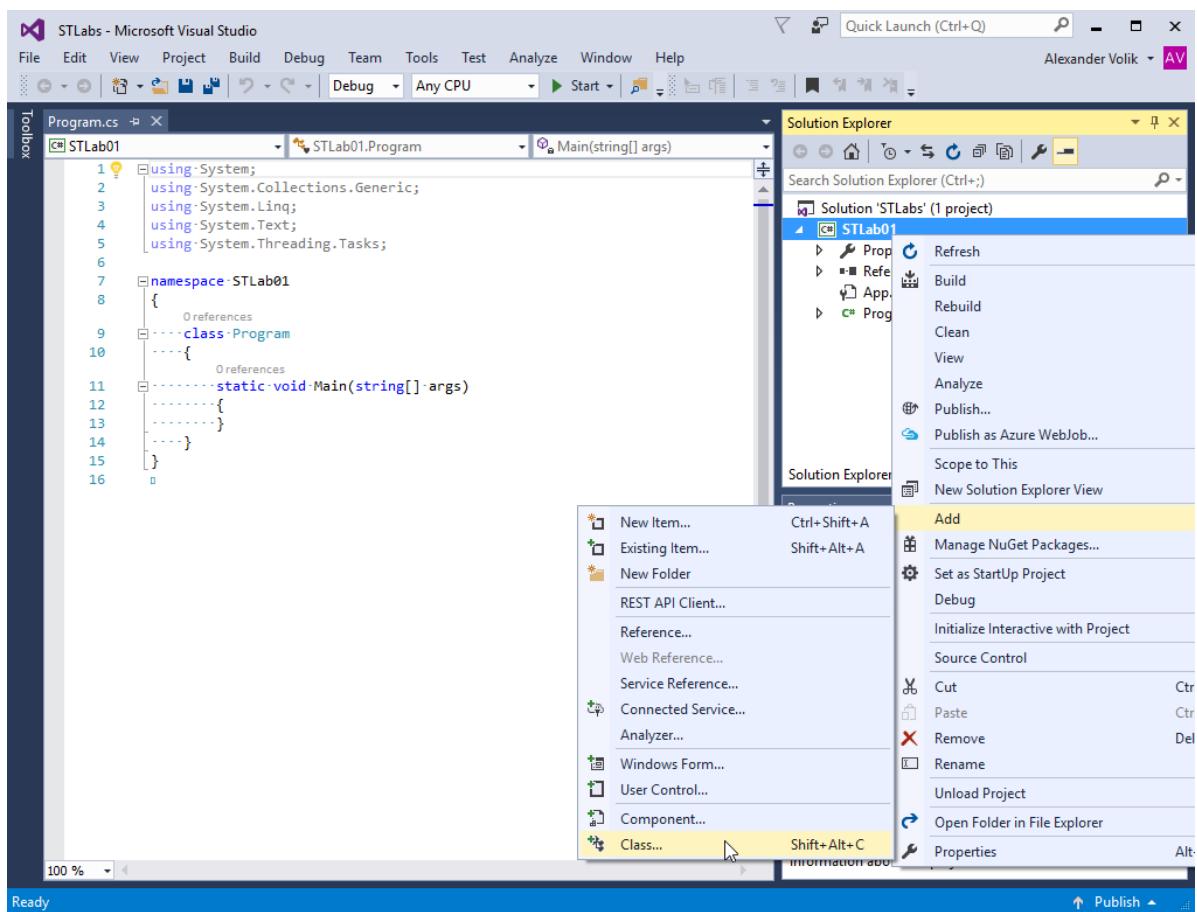


Рисунок Б.22 – Добавление класса

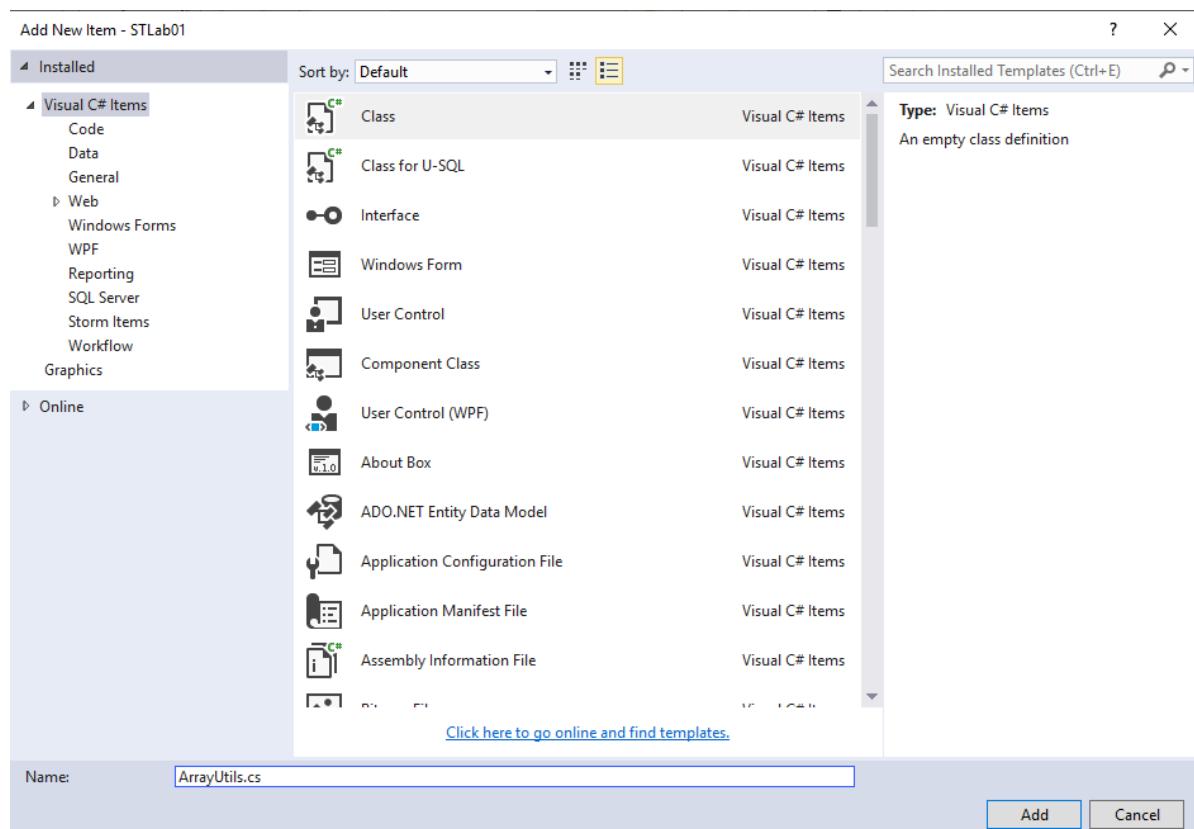


Рисунок Б.23 – Добавление нового класса

Далее реализуем в нем основные методы, приведённые в листинге Б.1.

Добавим новый тестовый проект в решение (рисунок Б.24). В этом случае создается новый пустой проект библиотеки классов без учета содержимого текущего проекта. Далее необходимо ввести имя нового тестового проекта. Имя выбирается в соответствии со следующими соглашениями: к имени проекта добавляется слово «Test» (рисунок Б.25).

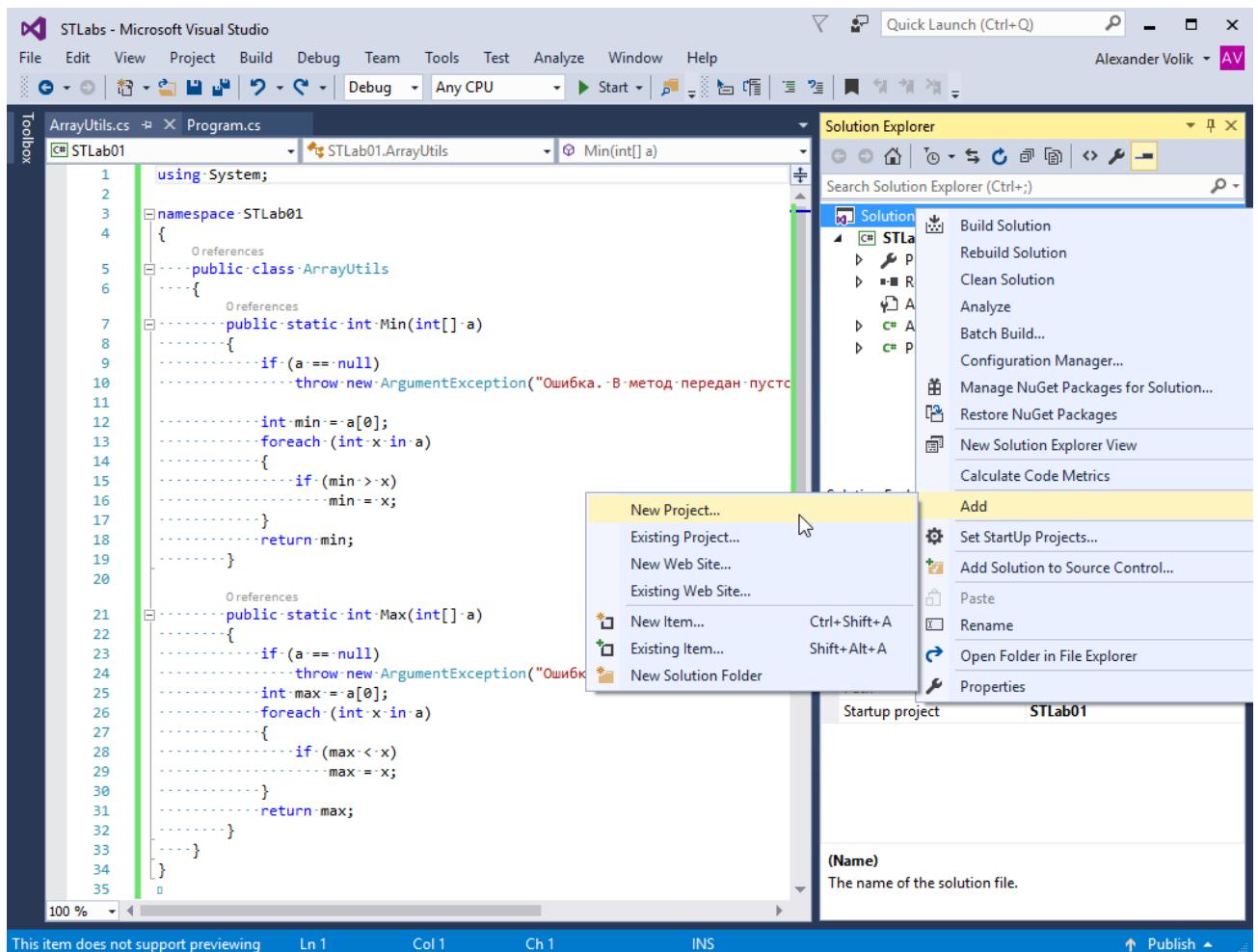


Рисунок Б.24 – Добавление нового тестового проекта

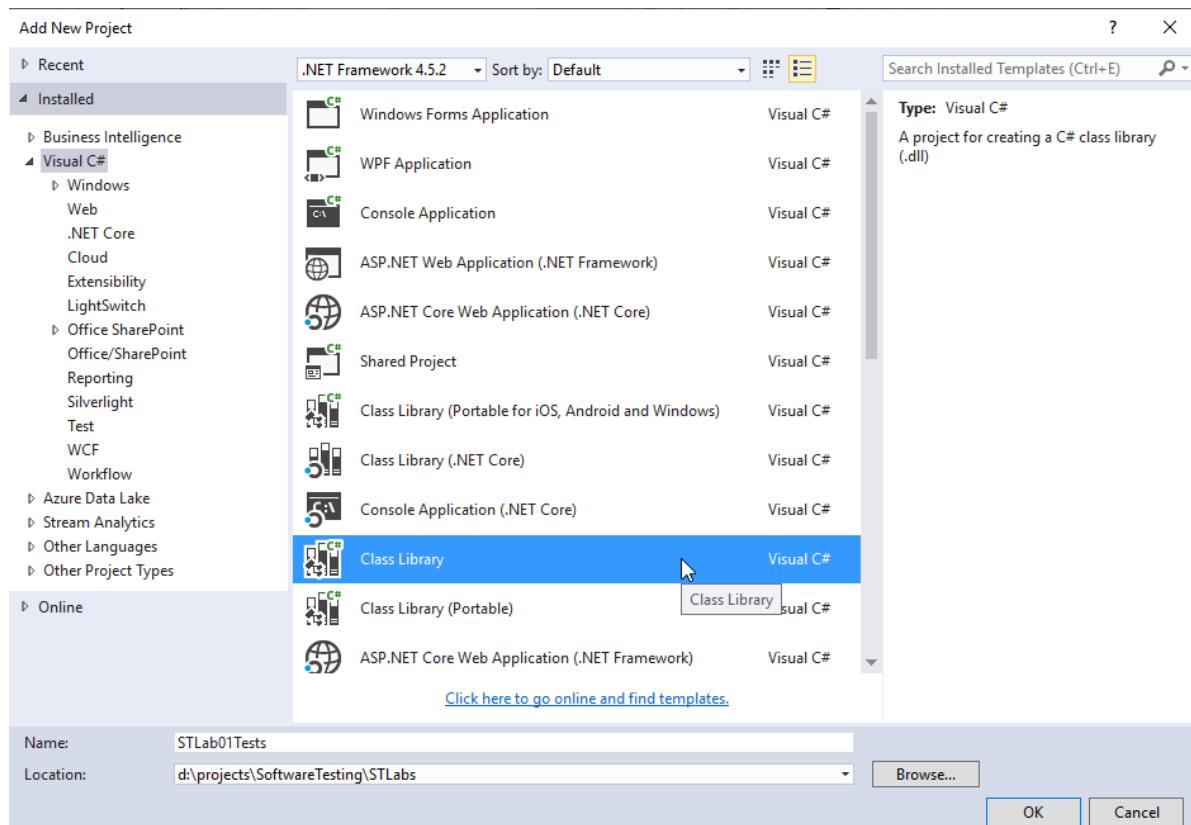


Рисунок Б.25 – Настройка тестового проекта

После этого Необходимо поменять имя файла тестов и содержащегося в нем класса, в соответствии с принятыми соглашениями: к имени тестируемого класса необходимо добавить слово Test. При этом для каждого класса, подвергаемого тестированию, в тестовом проекте создается файл модульного теста.

Тестовый проект содержит файлы, необходимые для модульных тестов. Свойства тестового проекта содержат файл AssemblyInfo.cs, в котором содержатся параметры построения проекта.

Далее необходимо настроить ссылки на сборки NUnit для получения возможности использования фреймворка. Для этого необходимо подключить соответствующую сборку из nuget-пакета (рисунки Б.26-Б.28).

Помимо этого в среде Visual Studio необходимо также установка пакета «NUnit Test Application» в котором добавляется поддержка работы с фреймворком NUnit (рисунки Б.29).

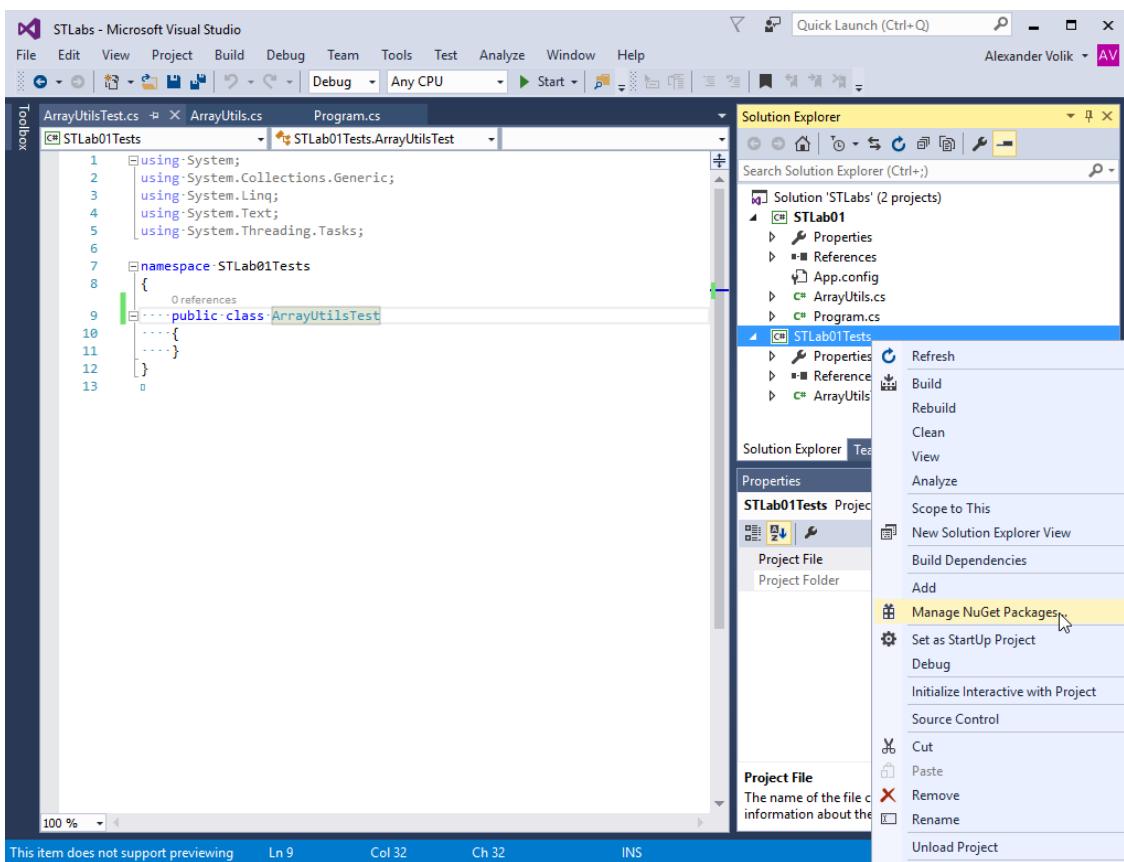


Рисунок Б.26 – Открытие вкладки управления nuget-пакетами

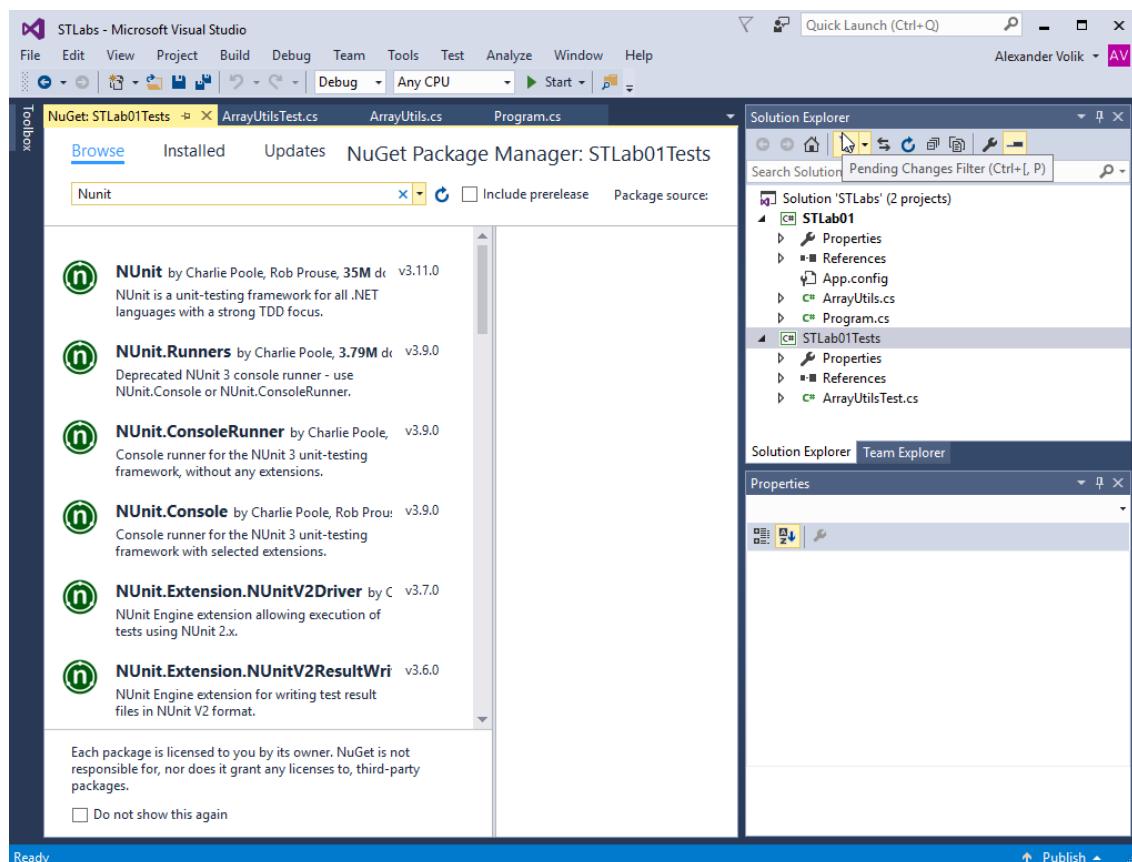


Рисунок Б.27 – Поиск nuget-пакета

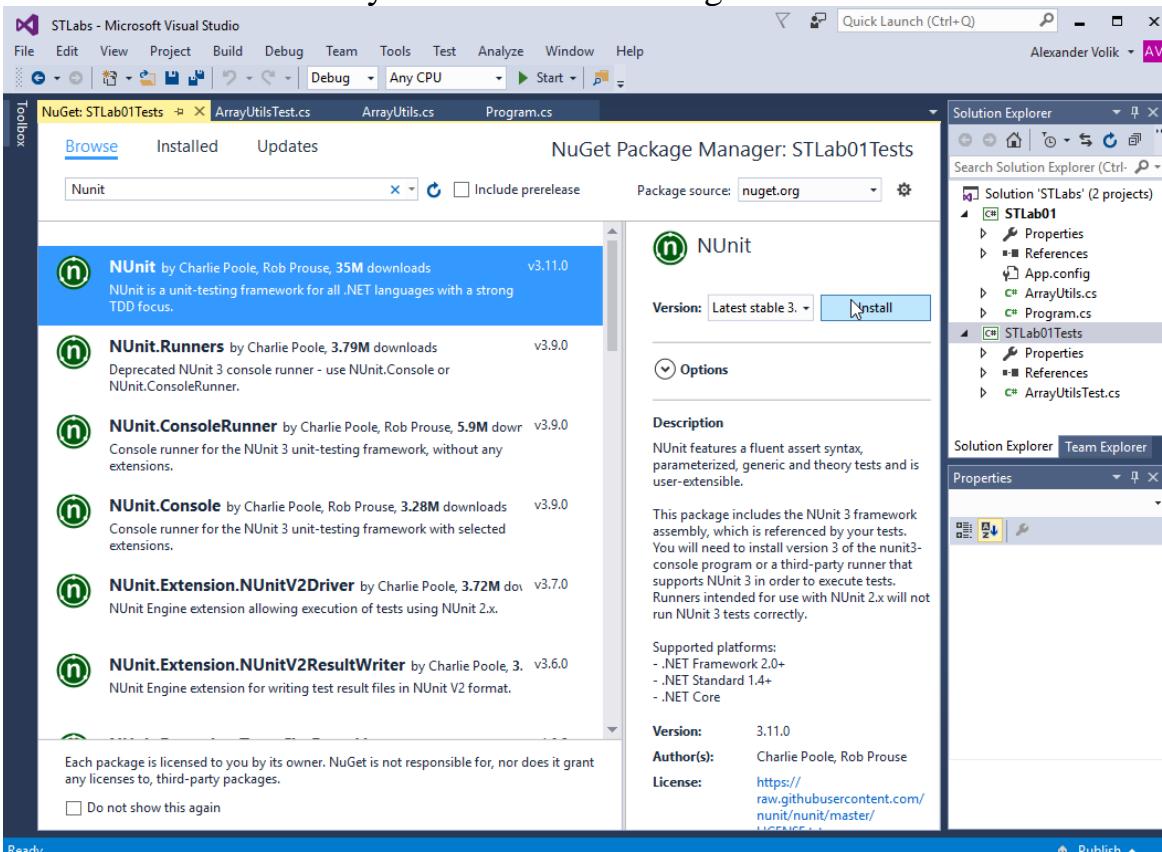


Рисунок Б.28 – Добавление nuget-пакета Nunit в проект

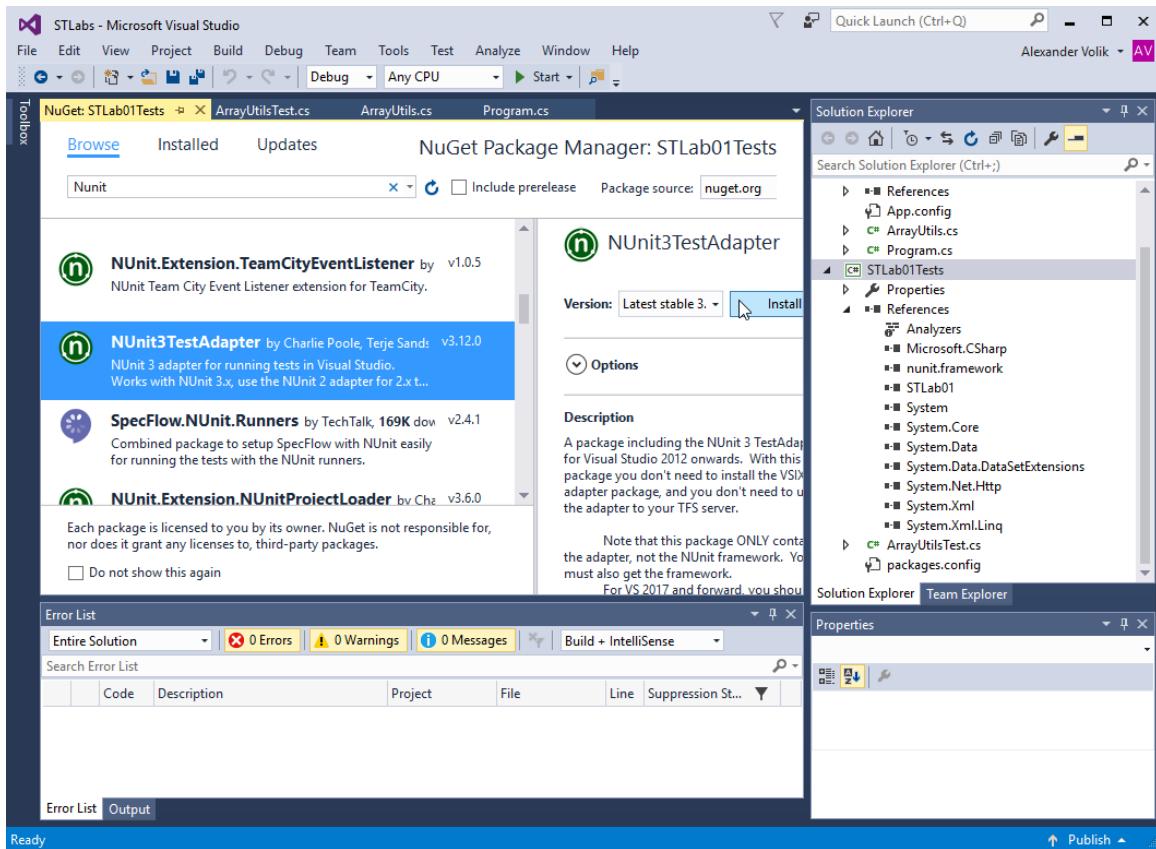


Рисунок Б.29 – Добавление nuget-пакета Nunit Test Adapter

Кроме этого необходимо добавить ссылку на сборку тестируемого проекта (рисунки Б.30- Б.32).

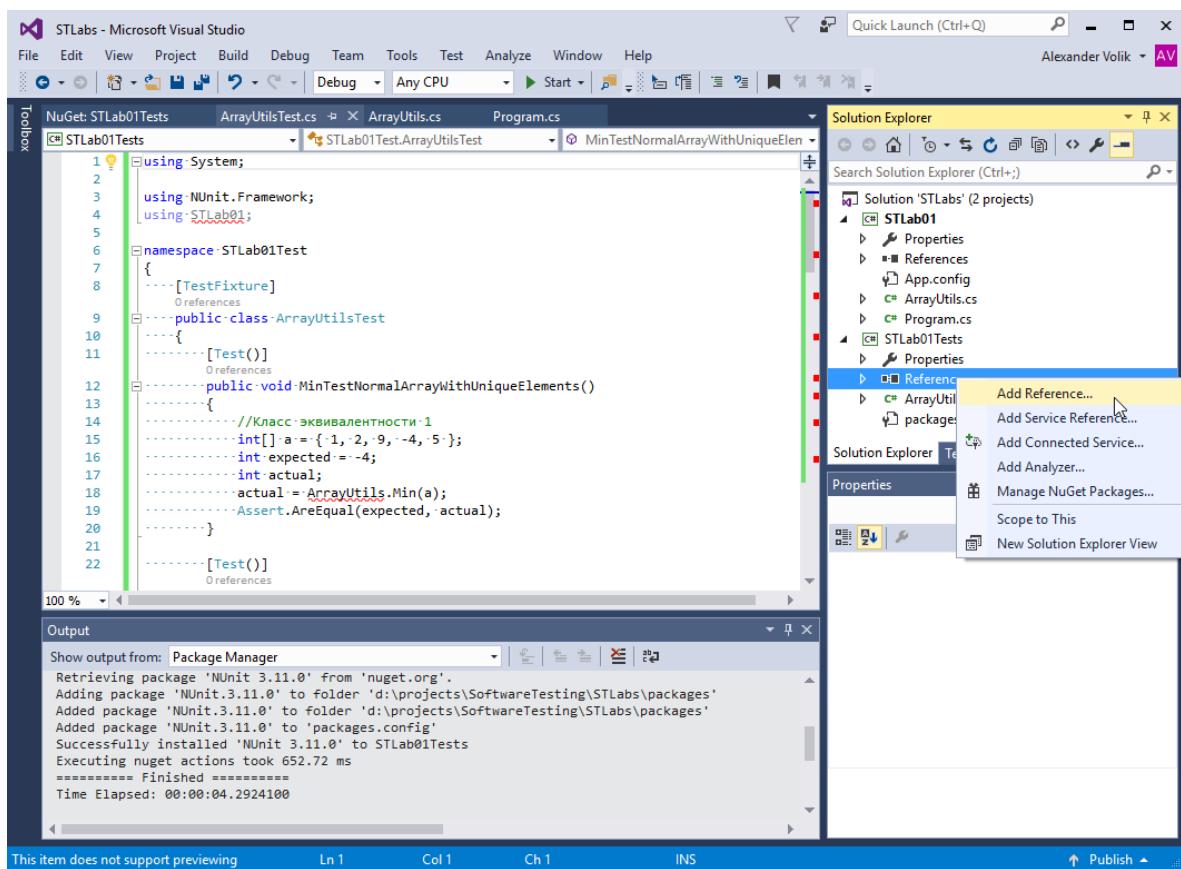


Рисунок Б.30 – Открытие окна добавления ссылки на проект

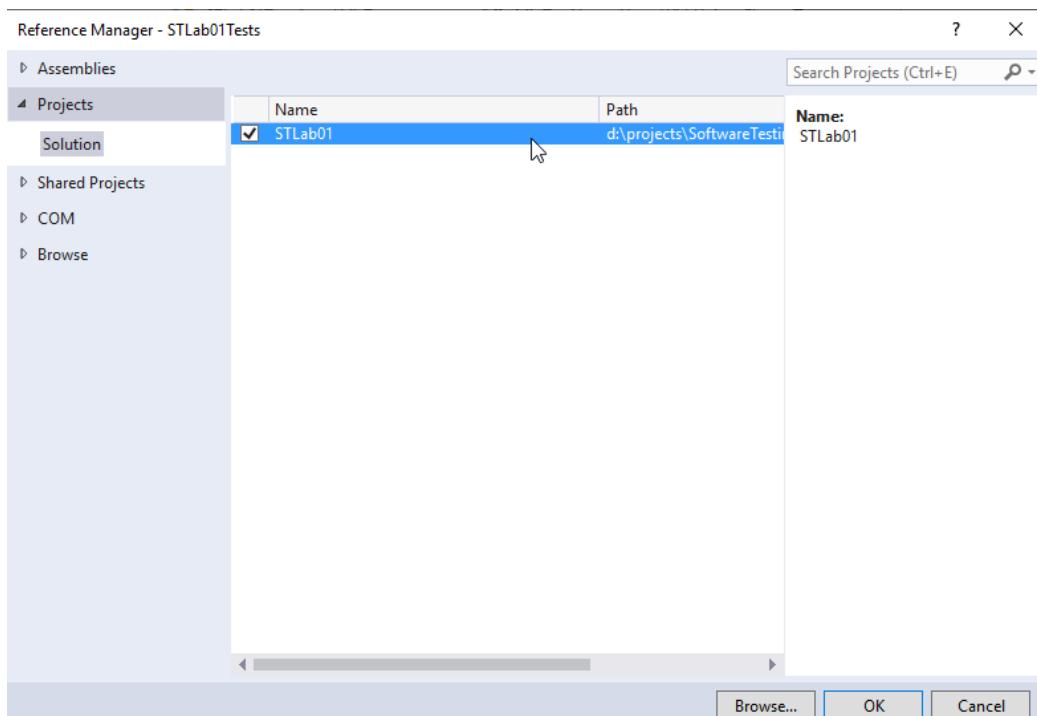


Рисунок Б.31 – Добавление сборки основного проекта

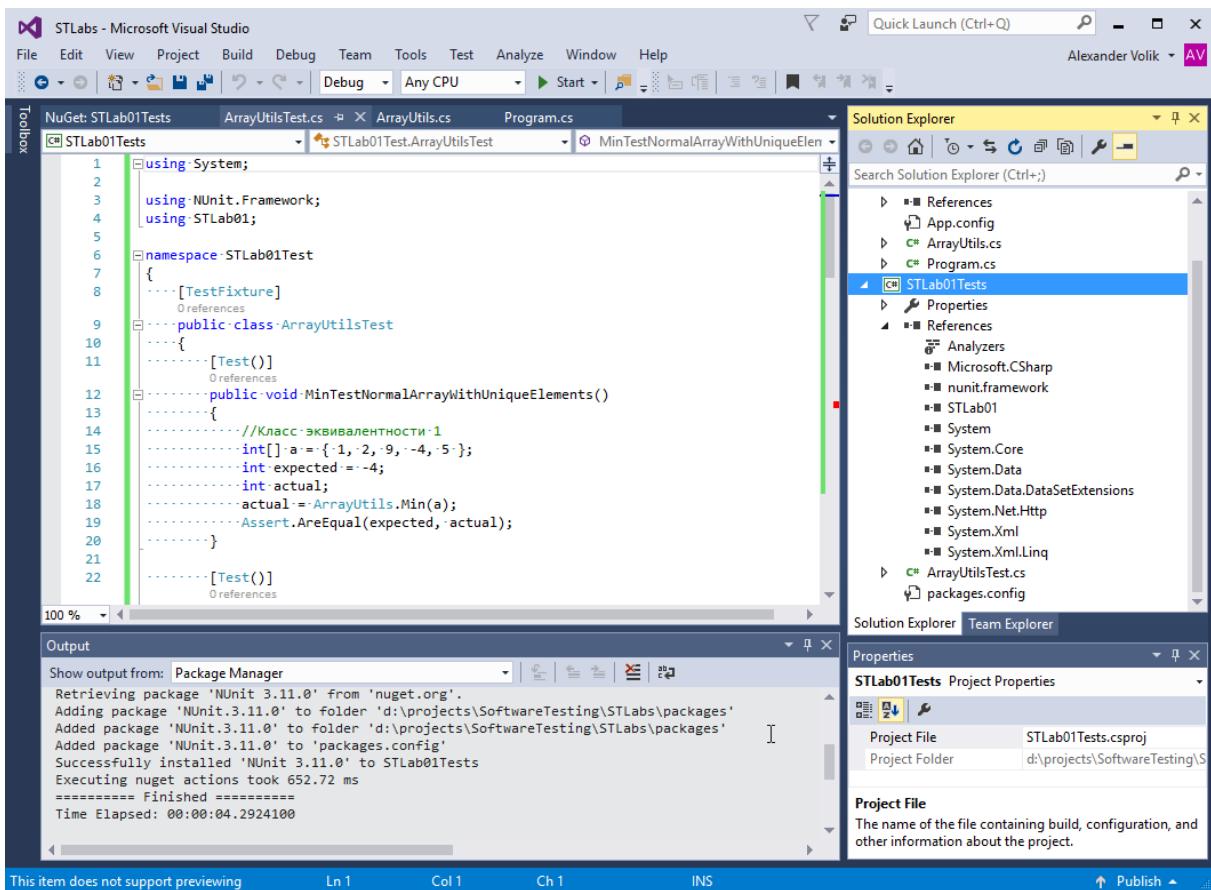


Рисунок Б.32 – Результат добавления сборки тестируемого проекта

Далее в коде тестового класса необходимо реализовать тестовые методы из примера Б.2.

После установки плагина «NUnit Test Application» в среду Visual Studio добавляется поддержка работы с фреймворком NUnit. Для того чтобы увидеть наборы тестов для выполнения необходимо обновить содержимое окна Test Explorer (рисунок Б.33). Оно открывается через пункт меню Test => Windows => Test Explorer.

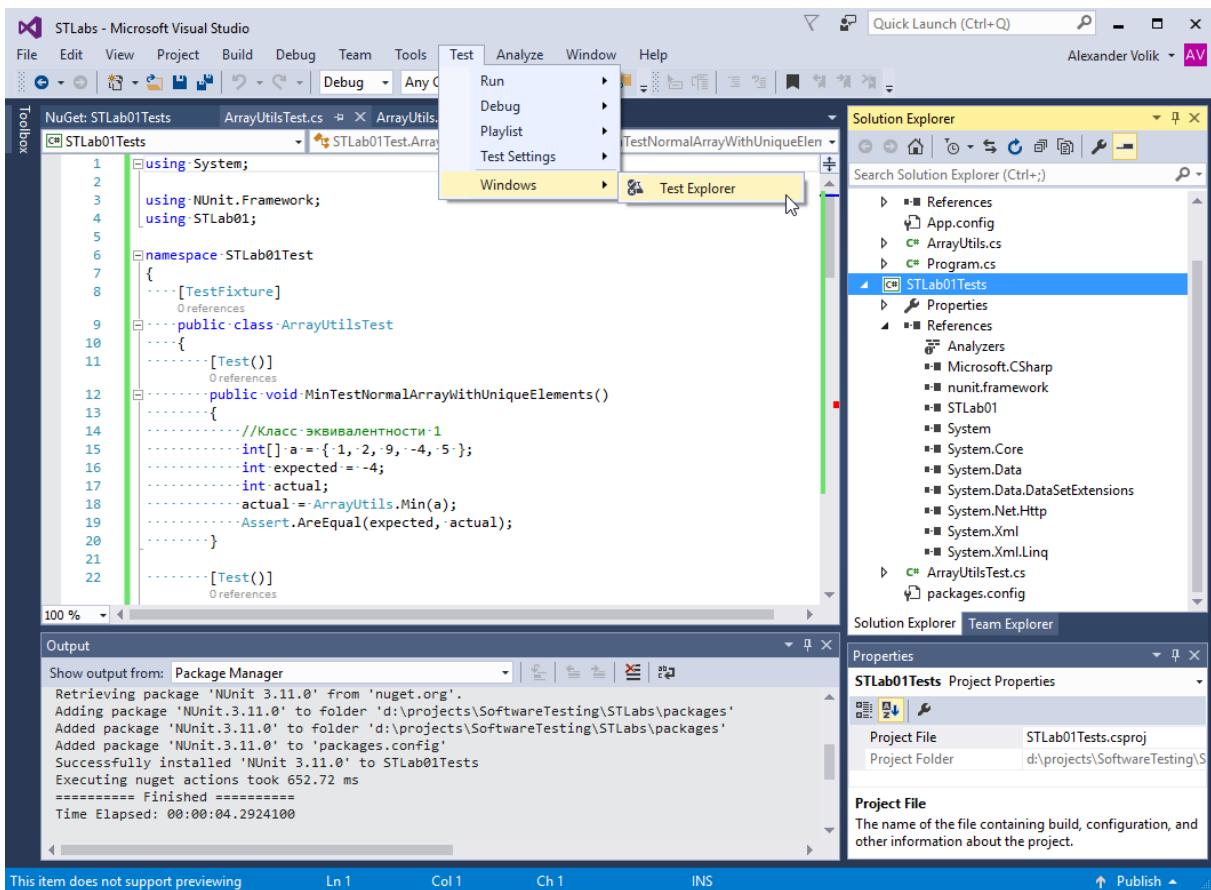


Рисунок Б.33 – Окно Test Explorer

После этого запустив набор созданных тестов (касающихся метода `Min`) будет пройдена, а часть останется (для метода `Max`) (рисунок Б.34).

После успешного прохождения теста он исключается из списка для дальнейших запусков до внесения изменений в код.

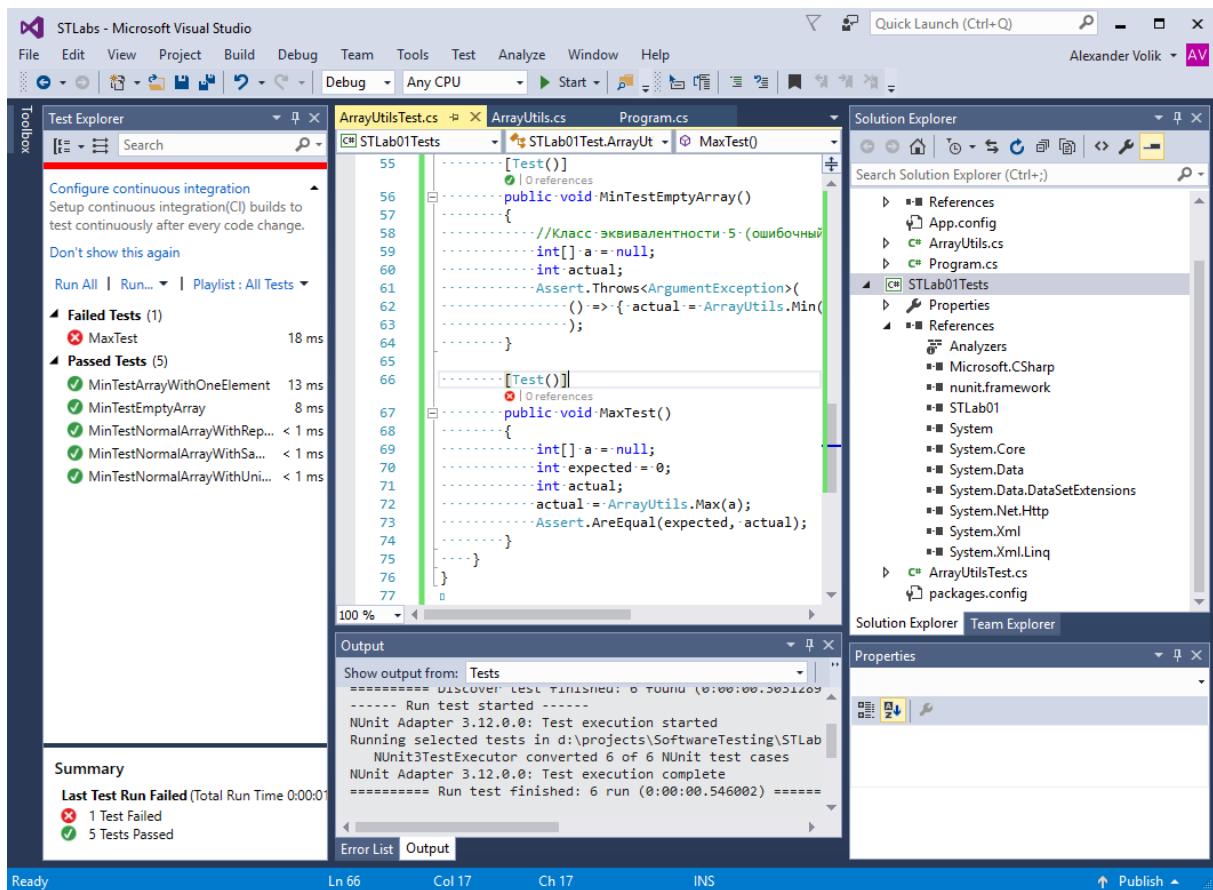


Рисунок Б.34 – Частичное прихождение тестов

Допишем оставшиеся тесты для метода Max из примера Б.3. В итоге мы получим полностью проходящий тесты класс (рисунки Б.35 и Б.36) и теперь привнесении любых изменений сможем определить, не внесли ли мы какую либо ошибку, просто запустив тесты заново.

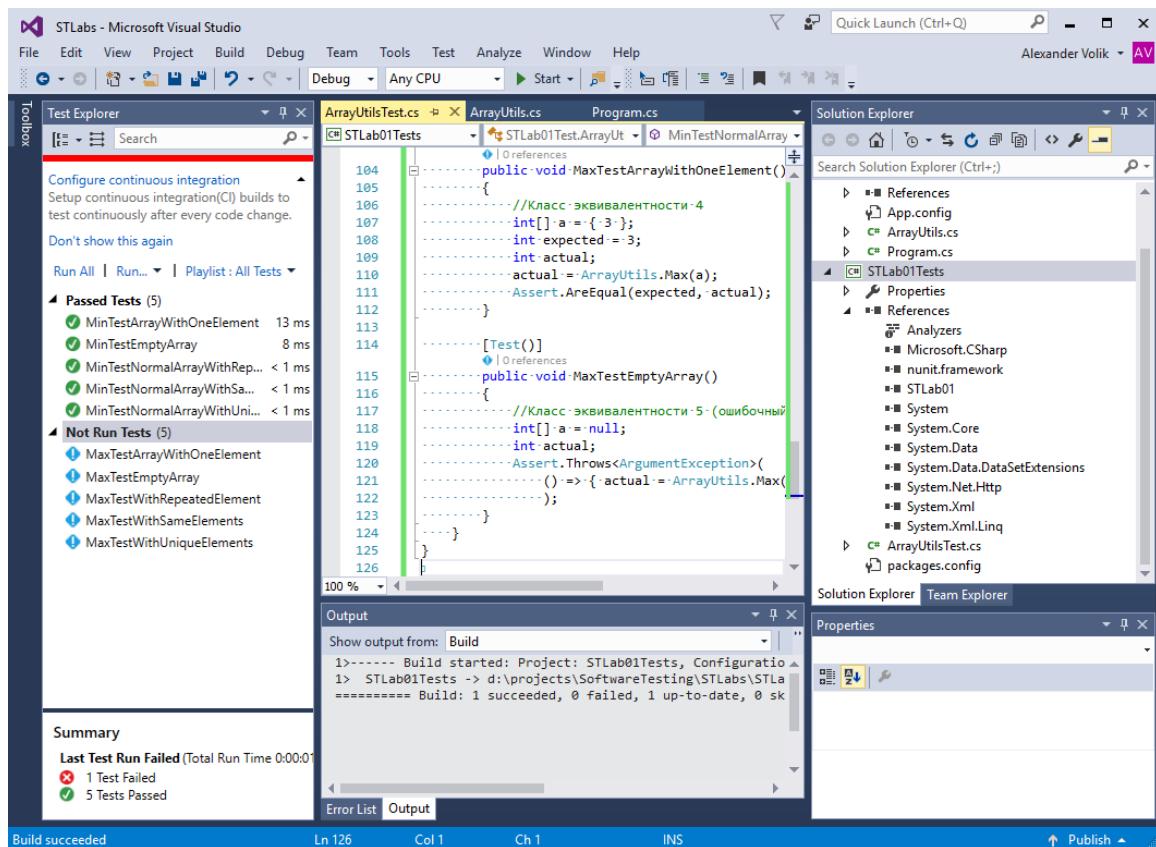


Рисунок Б.35 – Окно Test Explorer после пересборки проекта

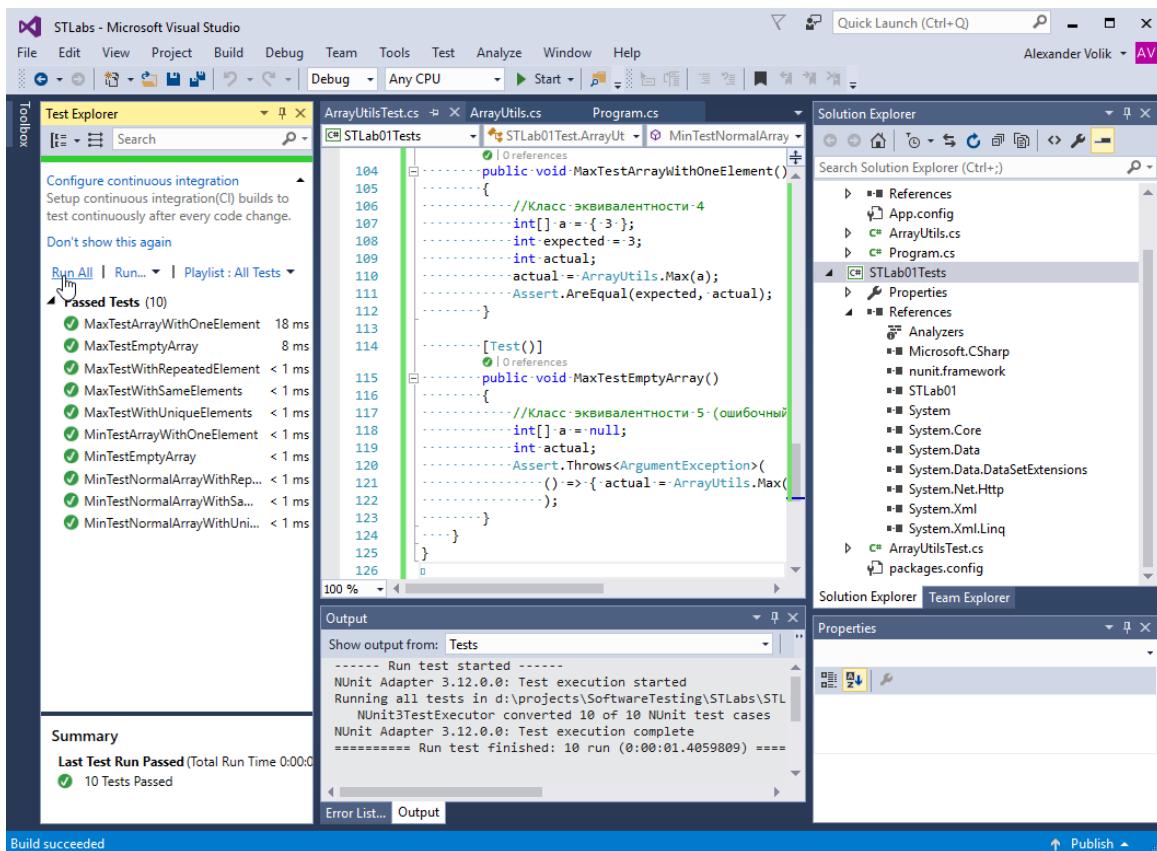


Рисунок Б.36 – Полное приходжение тестов

Б.7.3 Особенности создания проектов .NET Core

После появления кроссплатформенного решения .NET Core появилось несколько типов проектов (рисунки Б.37 и Б.38), которые не совместимы между собой. Классические windows приложения, созданные для .Net Framework (4.5/4.6/4.7/4.8) не могут нормально работать с библиотеками, написанными для .NET Core и наоборот. Поэтому при создании проекта приложения и тестового проекта для него необходимо учитывать вариант фреймворка и выбирать только один из них: для Console App (.NET Core) нужно выбирать Class Library (.NET Core), а для Console App (.Net Framework) нужно выбирать Class Library (.Net Framework).

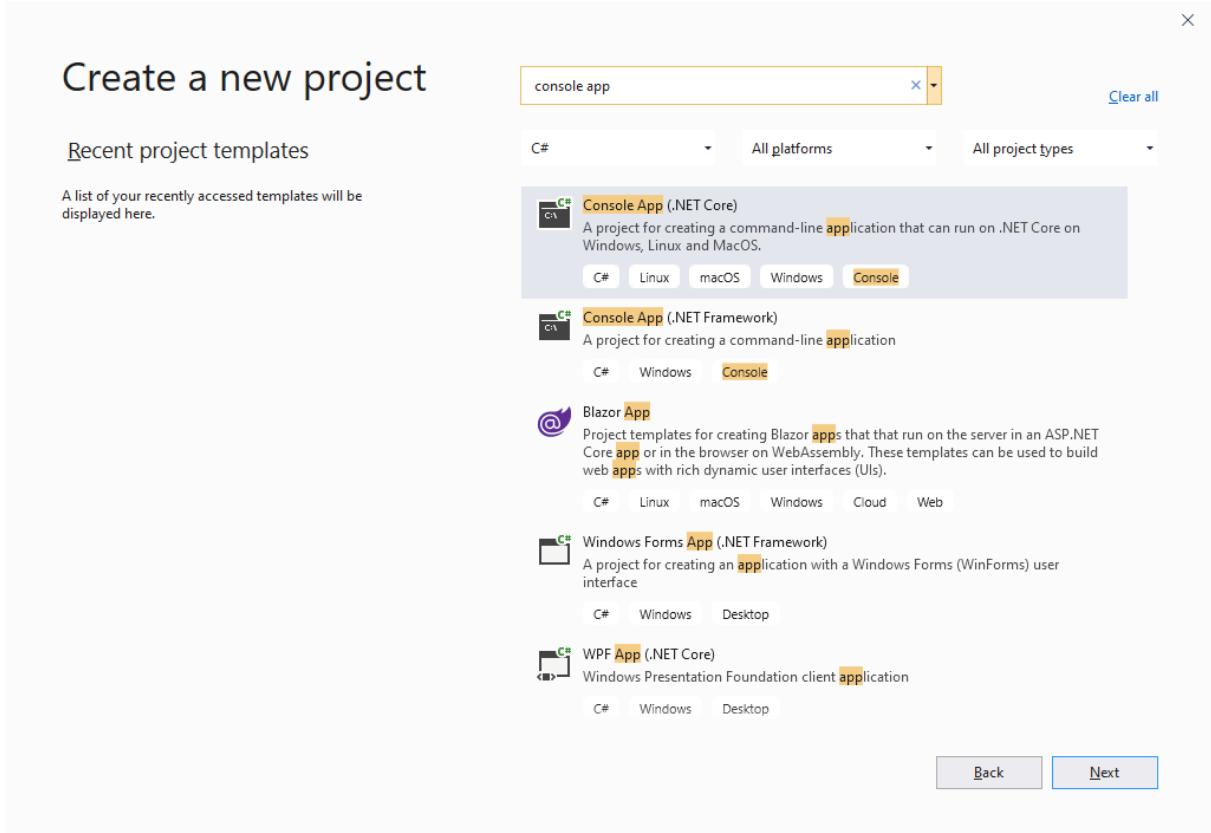


Рисунок Б.37 – Варианты консольных приложений (Console App)

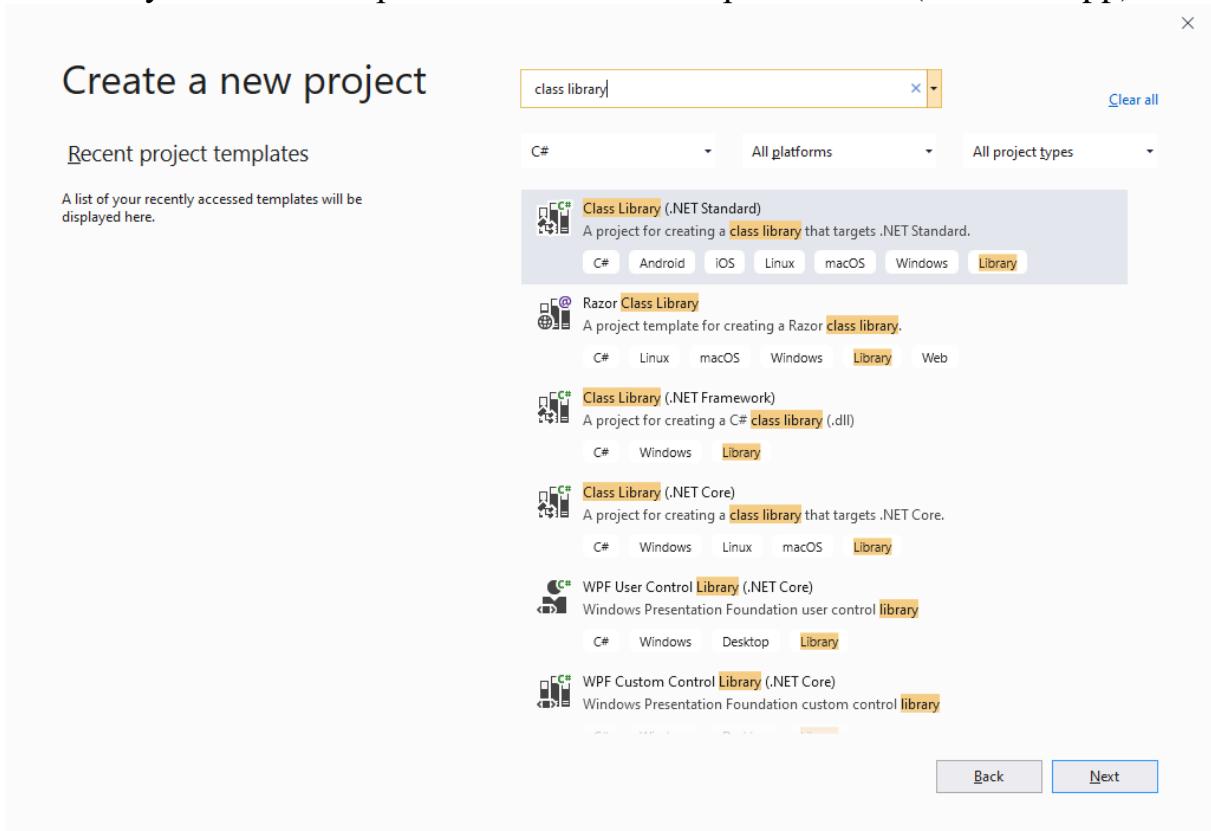


Рисунок Б.37 – Варианты библиотек классов (Class Library)

Помимо этого для запуска тестов из библиотеки .NET Core необходимо установить еще один дополнительный пакет (помимо уже установленных NUnit и NUnitTestAdapter): **Microsoft.NET.Test.Sdk** (рисунок Б.38).

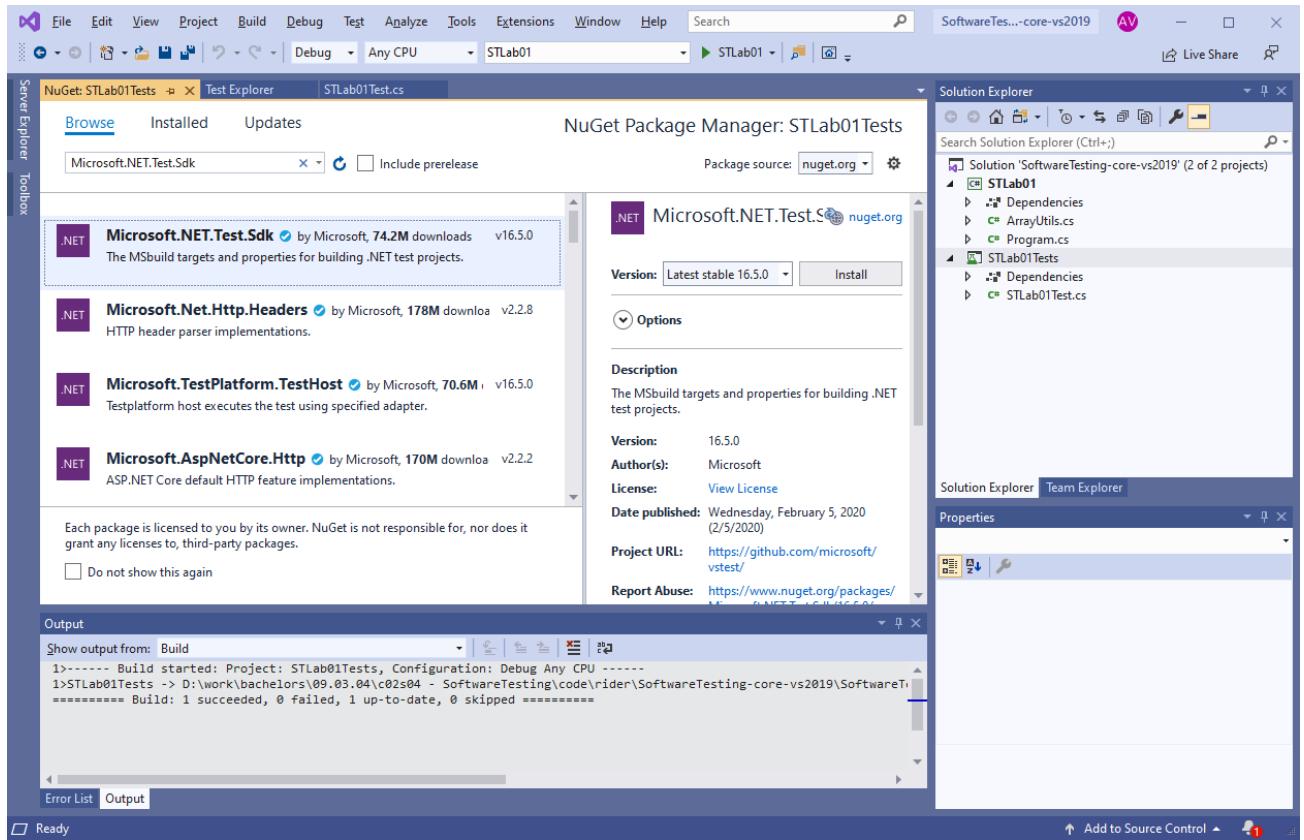


Рисунок Б.38 – Установка дополнительного пакета Microsoft.NET.Test.Sdk для запуск тестов .NET Core из Visual Studio

Приложение С (справочное)

Работа с системой управления пакетами NuGet

C.1 Система управления пакетами NuGet

До появления платформы .Net стояла очень остро проблема, которая называлась DLL Hell. С появлением .Net и возможности непосредственно в сборке хранить информацию о версии, подписи и т.д. Проблема DLL Hell ушла в прошлое, GAC позволяет хранить в одном месте сборки разных версий. Но, следуя тем же рекомендациям от Microsoft, если сборки не предполагаются для глобального использования, то лучше их размещать не в GAC, а размещать вместе с приложением. Но в этом случае возникает проблема, как подключать эти библиотеки в решения, как поддерживать их актуальную версию, особенно, если библиотеки, являются внешними по отношению к организации.

Для решения этой и многих других проблем, было разработано расширение для Visual Studio, которое получило название NuGet.

NuGet – система управления пакетами для платформ разработки Microsoft, в первую очередь библиотек .NET Framework. Основной задачей NuGet является упрощение процесса установки сторонних библиотек.

Он берет на себя все шаги этого процесса, в том числе:

- поиск библиотеки;
- загрузка необходимого для установки пакета;
- проверка его хэш-значения на соответствие с заданным сервером (проверка целостности);
- распаковка файлов библиотеки в нужное место;
- добавление ссылок (reference) на её сборки в проект;
- модификацию файлов конфигурации (web.config, app.config) при необходимости.

Причем учитываются все зависимости загружаемой библиотеки от других компонентов. Последние, при необходимости, также будут загружены и добавлены в проект.

NuGet позволяет добавлять, обновлять и удалять библиотеки в виде пакетов. NuGet-пакет является набором файлов, упакованных в один файл с расширением .nupkg.

Информация о пакетах получается через фиды. По умолчанию NuGet автоматически подписывается на официальный фид NuGet. Так же можно добавлять дополнительные фиды, а так же использовать персональные фиды для внутреннего использования, которые могут находиться на сетевом диске.

C.2 Установка NuGet

В Visual Studio начиная с 2012 NuGet Manager уже установлен по умолчанию

C.3 Установка пакетов с помощью NuGet в Visual Studio

Для установки пактов можно использовать Графический менеджер пакетов (Package Manager UI) или Консольный менеджер пактов (Package Manager Console).

C.3.1 Графический менеджер пакетов (Package Manager UI)

Для начала в используемом проекте необходимо открыть контекстное меню на проекте (или решении) (рисунок C.1).

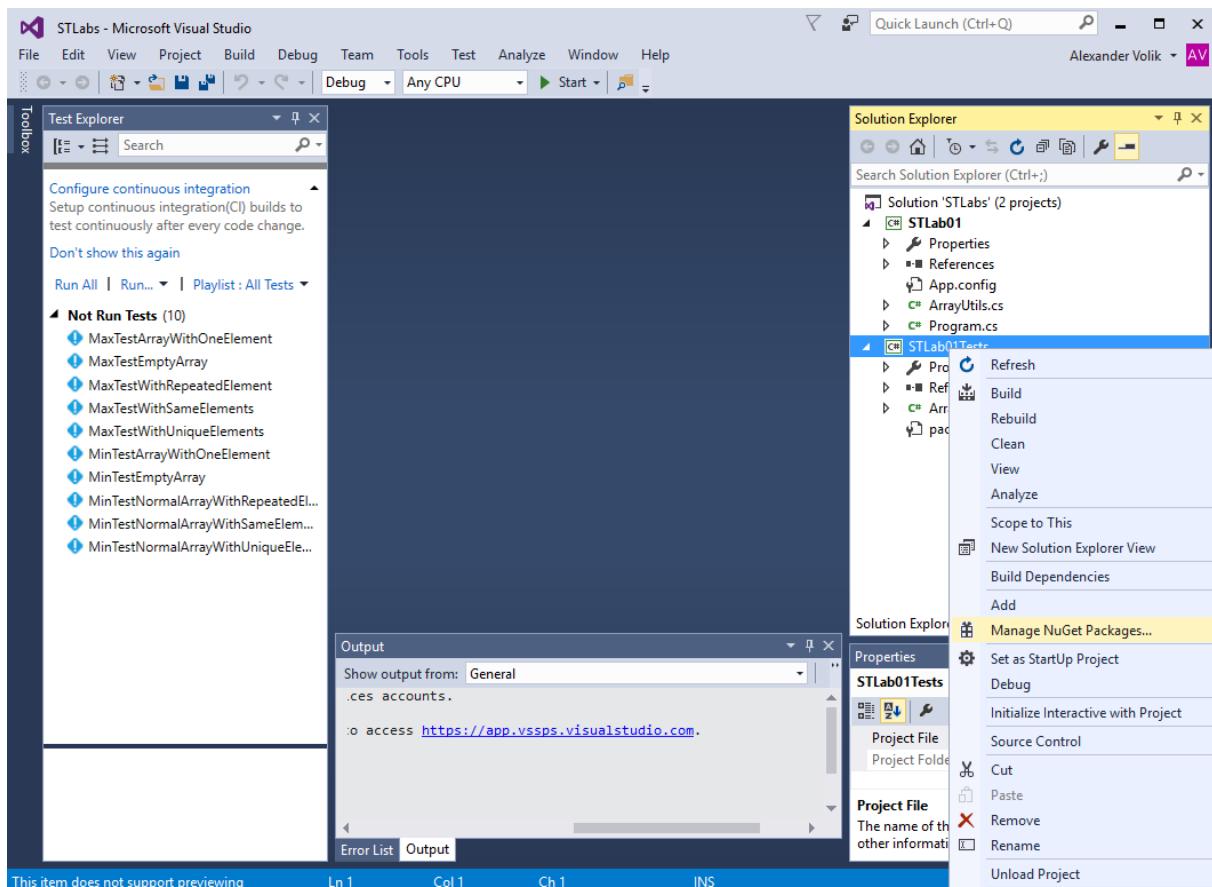


Рисунок С.1 – Вызов контекстного меню

При выборе Manage NuGet Packages на экране появляется окно, представленное на рисунке С.2.

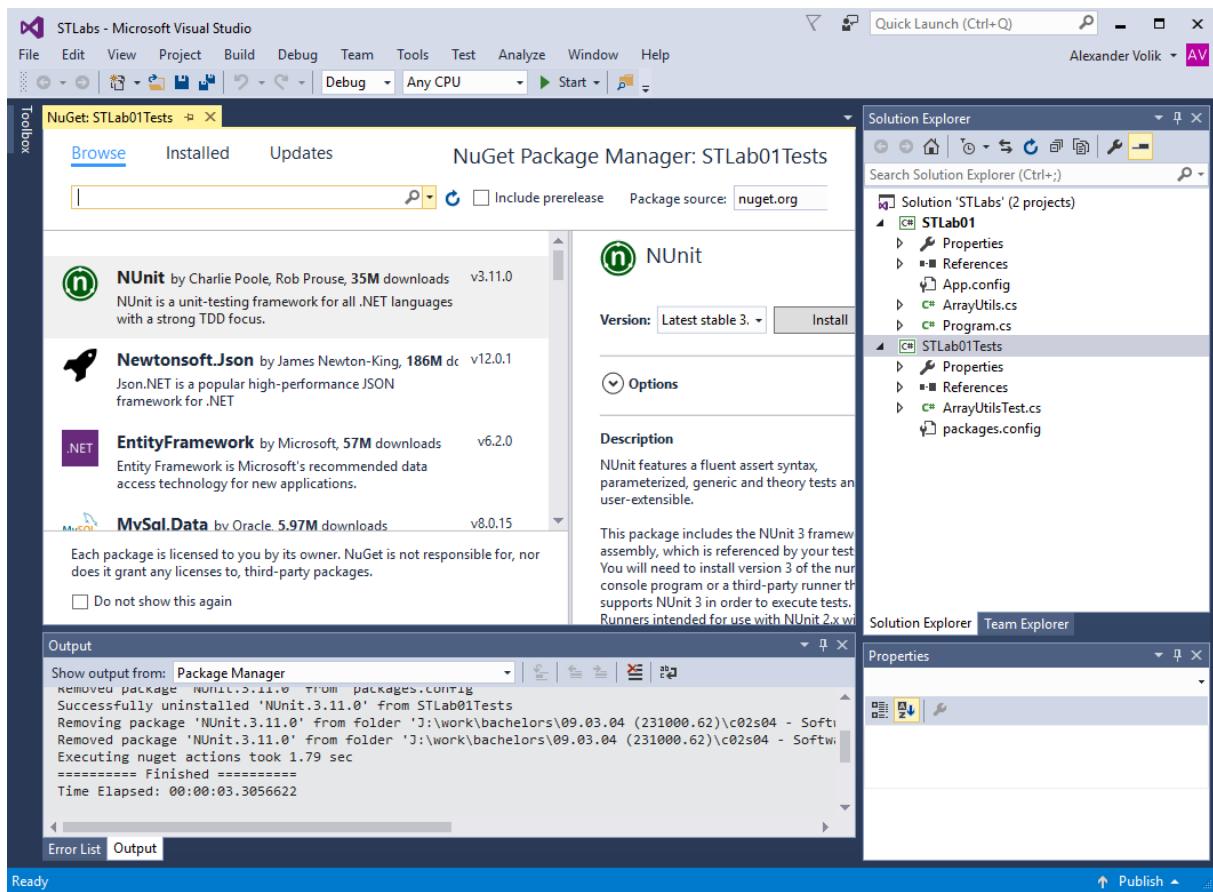


Рисунок С.2 – Окно менеджера пакетов

Для поиска и добавления пакета необходимо выбрать вкладку **Browse** (рисунок С.3), после чего в строку поиска вводим название искомого пакета, например **NUnit**. В загрузившемся списке выбираем требуемый пакет **NUnit** (рисунок С.4).

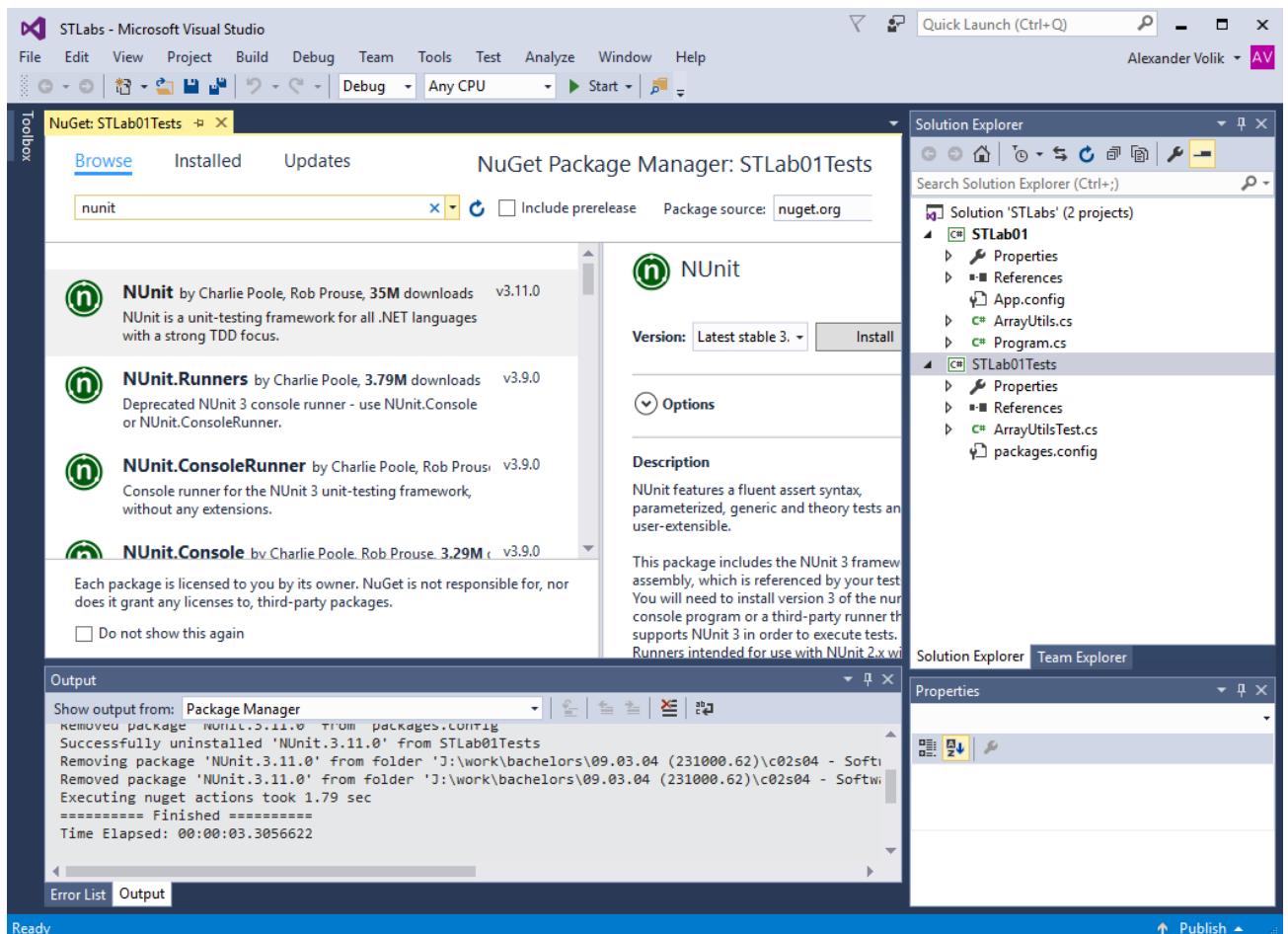


Рисунок С.3 – Вкладка Browse окна менеджера пакетов

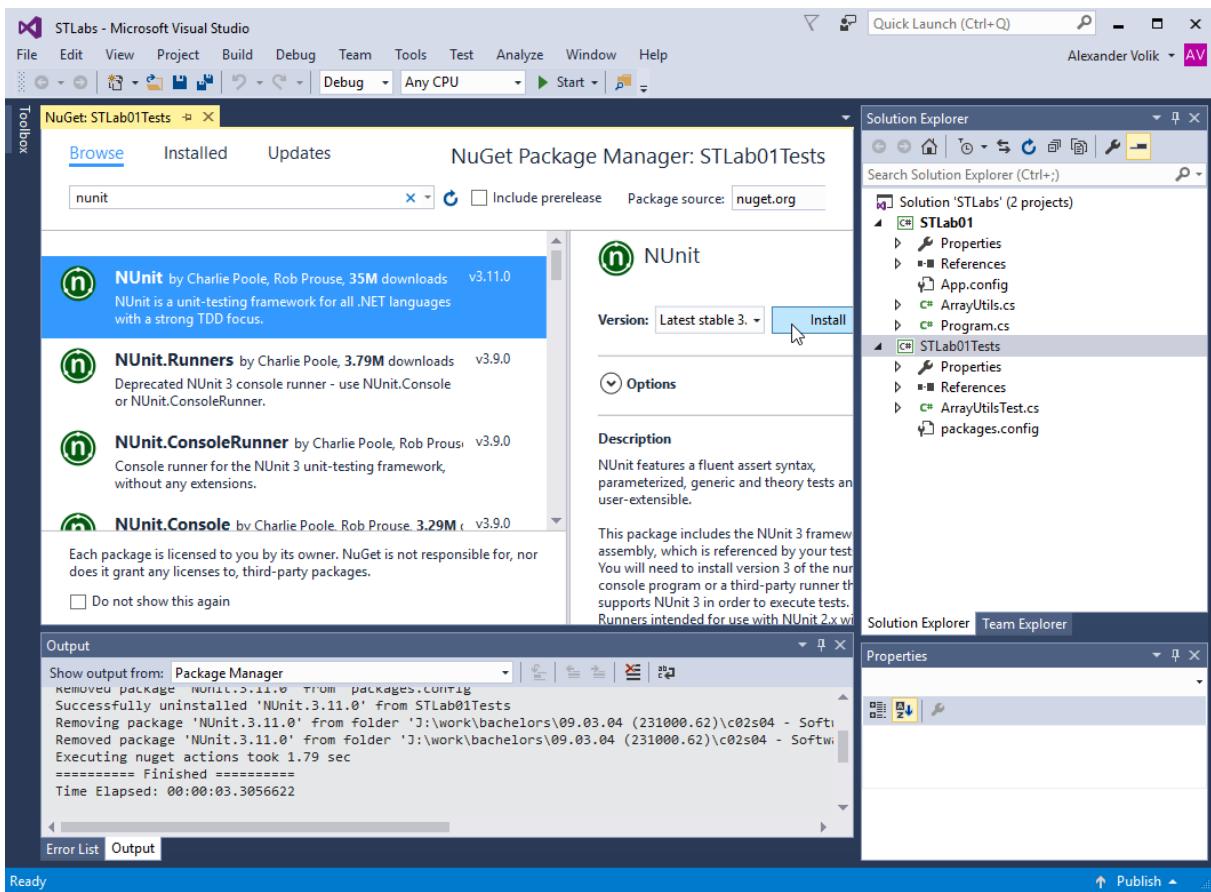


Рисунок С.4 – Поиск и установка пакетов

После установки решение примет вид, показанный на рисунке Рисунок С.5.

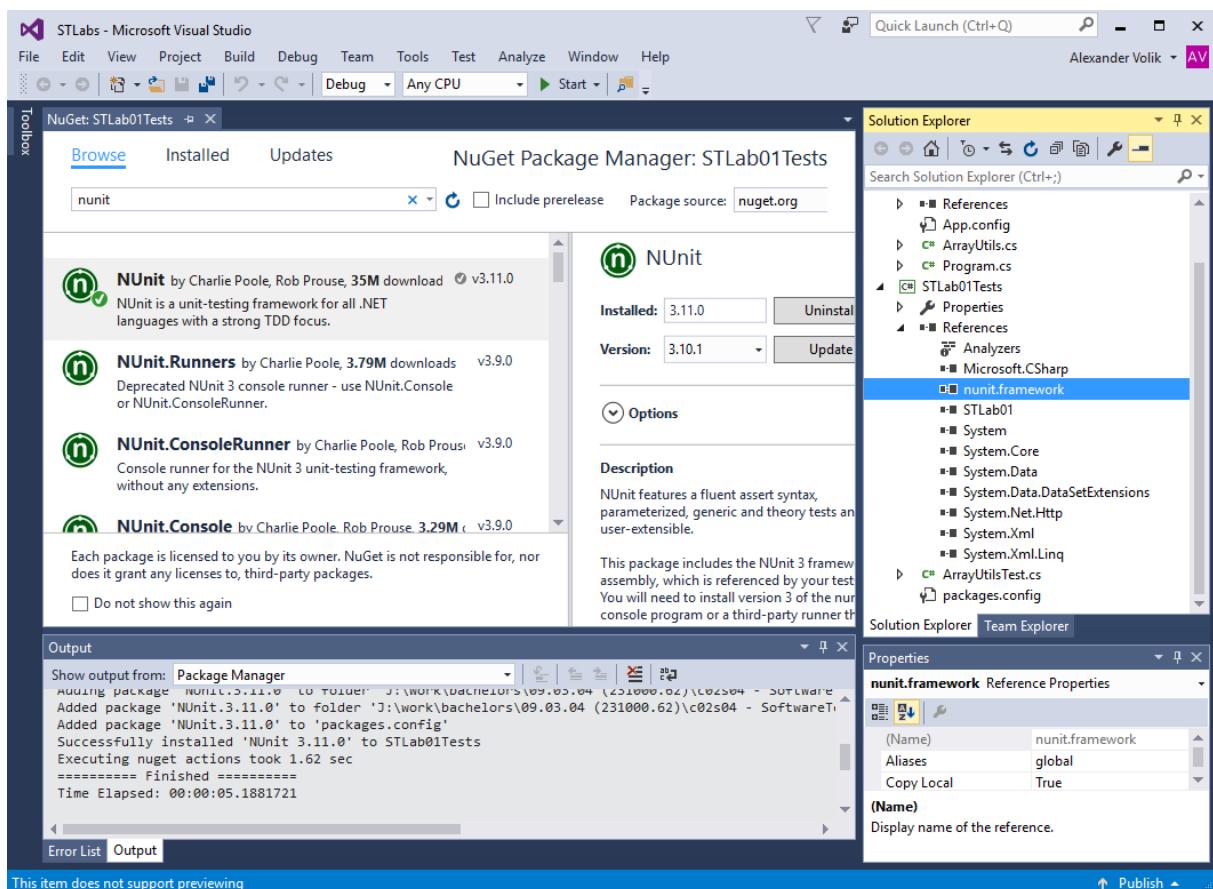


Рисунок С.5 – Состояние проекта после установки пакета

Теперь, после построения, в выходной папке будет не только само решение, но и dll поставляемые в пакетах.

Кроме этого NuGet создал в папке решения (solution) свою папку packages. По сути, это небольшой репозиторий проекта. Внутри него расположились DLL и остальные файлы загруженных библиотек. Причем структура размещения внутри каждой папки частично определяется требованиями NuGet, а частично самим установленным компонентом.

Кроме того, в проект добавился один файл "packages.config", который после операции содержит список используемых в текущем проекте пакетов.

Пример С.1 – Содержимое файла packages.config

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="NUnit" version="3.11.0" targetFramework="net452" />
</packages>
```

B.3.2 Консольный менеджер пактов (Package Manager Console)

Рассмотрим еще один способ работы с NuGet: Консольный менеджер пактов (Package Manager Console).

Package Manager Console представляет собой приложение, встроенное в Microsoft Visual Studio (начиная с версии 2012), позволяющее использовать NuGet PowerShell для поиска, установки и обновления пакетов.

Данный вариант подойдет тем, кому больше нравится работать с командной строкой, чем с диалоговыми окнами. Сразу необходимо отметить, что поддерживается подсказка для завершения набора команд. Для её активации необходимо ввести несколько первых букв и нажать кнопку "Tab".

Для начала откроем Package Manager Console с помощью пункта меню View -> Other Windows (рисунок С.6) или через пункт Tools -> Library Package Manager (рисунок С.7). После загрузки и инициализации PowerShell, который она использует, оно выглядит следующим образом:

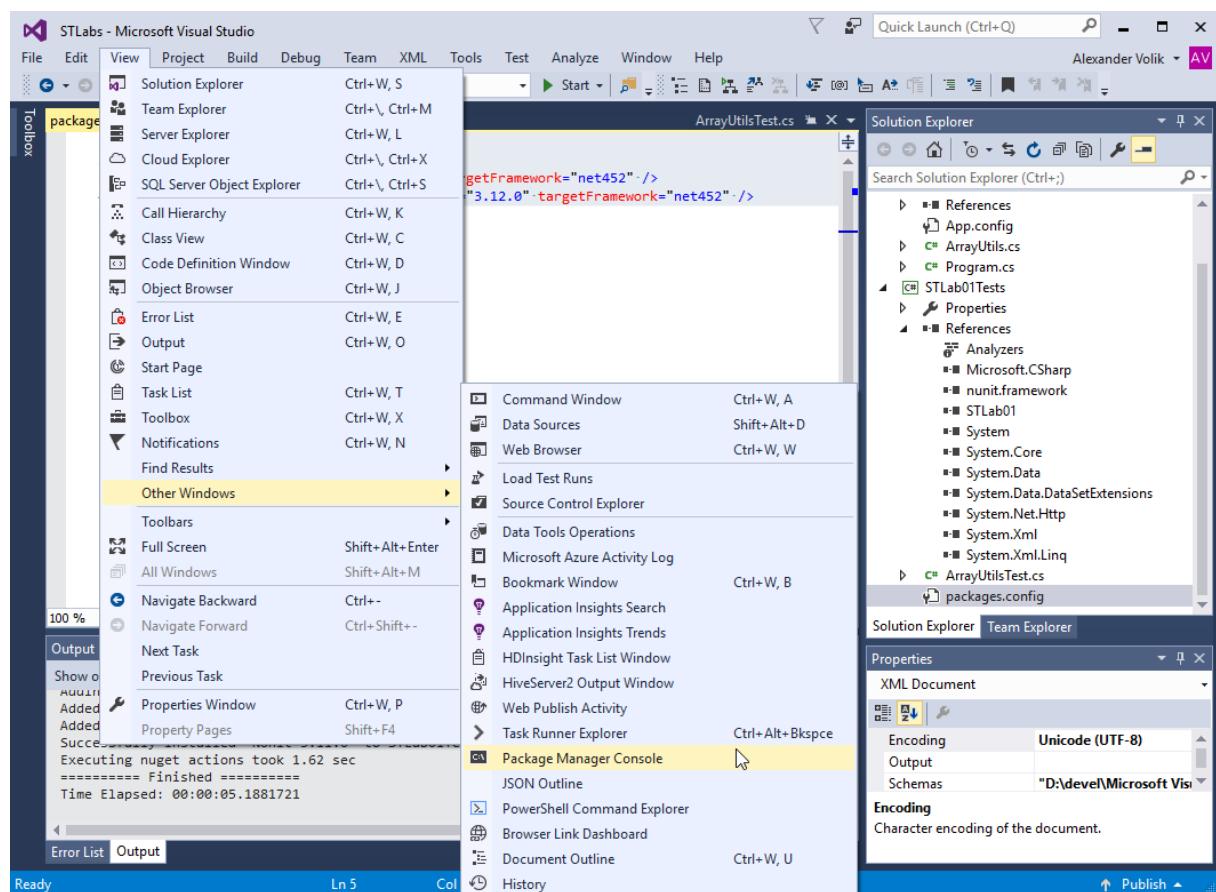


Рисунок С.6 – Открытие Package Manager Console через пункт View

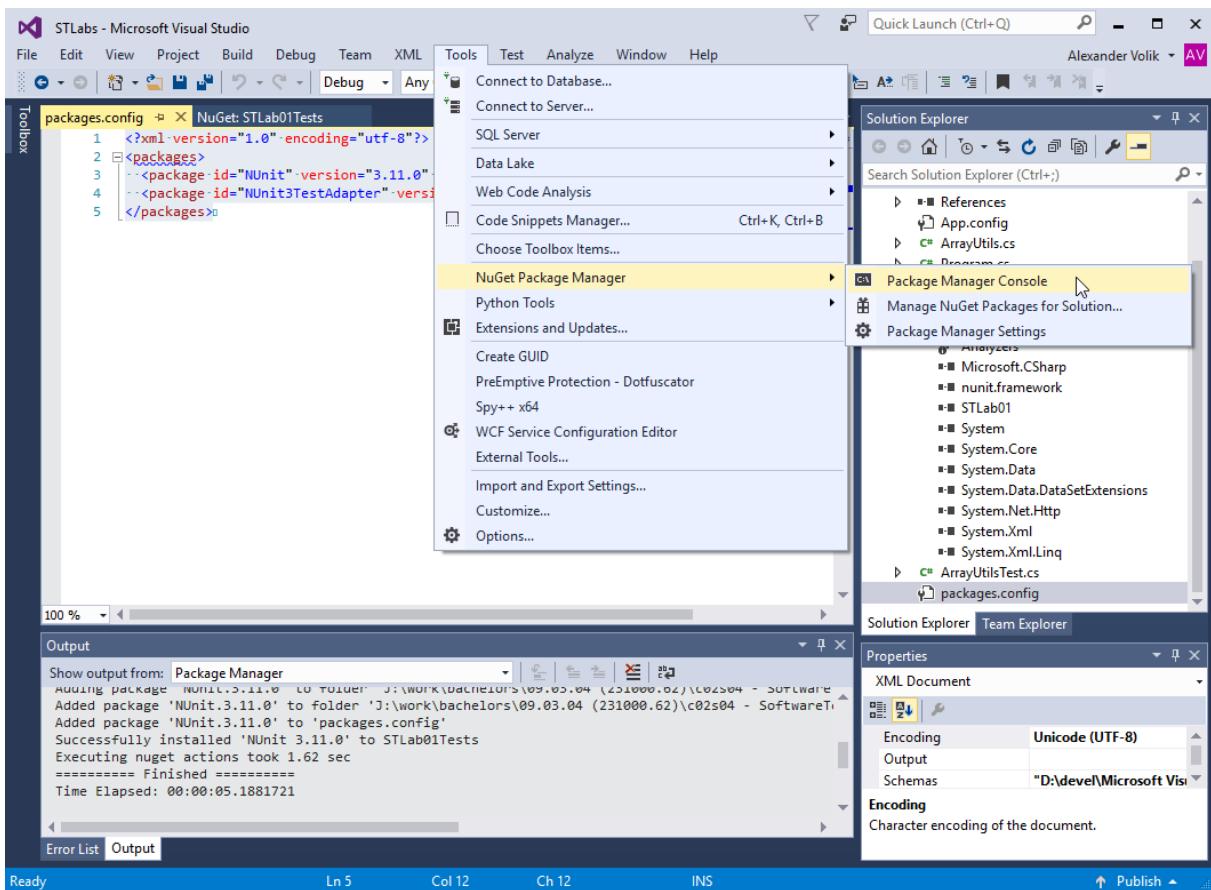


Рисунок С.7 – Открытие Package Manager Console через пункт Tools

После загрузки и инициализации PowerShell, который использует Package Manager Console, окно выглядит как представлено на рисунке С.8. После открытия консоли в ней выбран проект по умолчанию (выпадающий список в правом верхнем углу с подписью Default project).

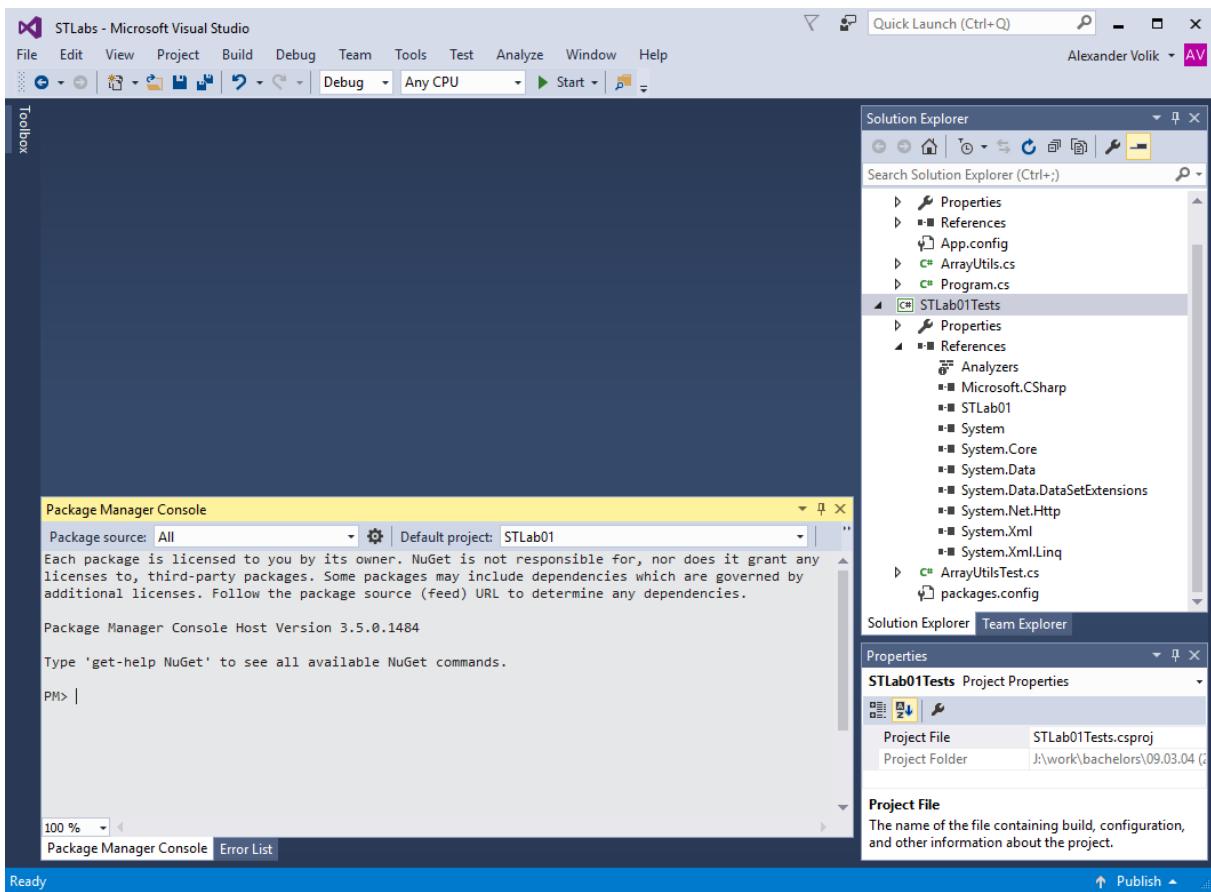


Рисунок C.8 – Окно Package Manager Console

Для начала рассмотрим, как выполнить поиск нужной библиотеки. Для этого необходимо дать команду "Find-Package" (более старые версии NuGet используют команду "Get-Package –ListAvailable"). Однако в этом случае будет выведен огромный список всех доступных установочных пакетов. Поэтому уточним команду, добавив строку для поиска. В итоге получится следующий вариант: "Find-Package NUnit". Результатом его работы будет список доступных библиотек, в имени или описании которых есть указанная строка (рисунок C.9).

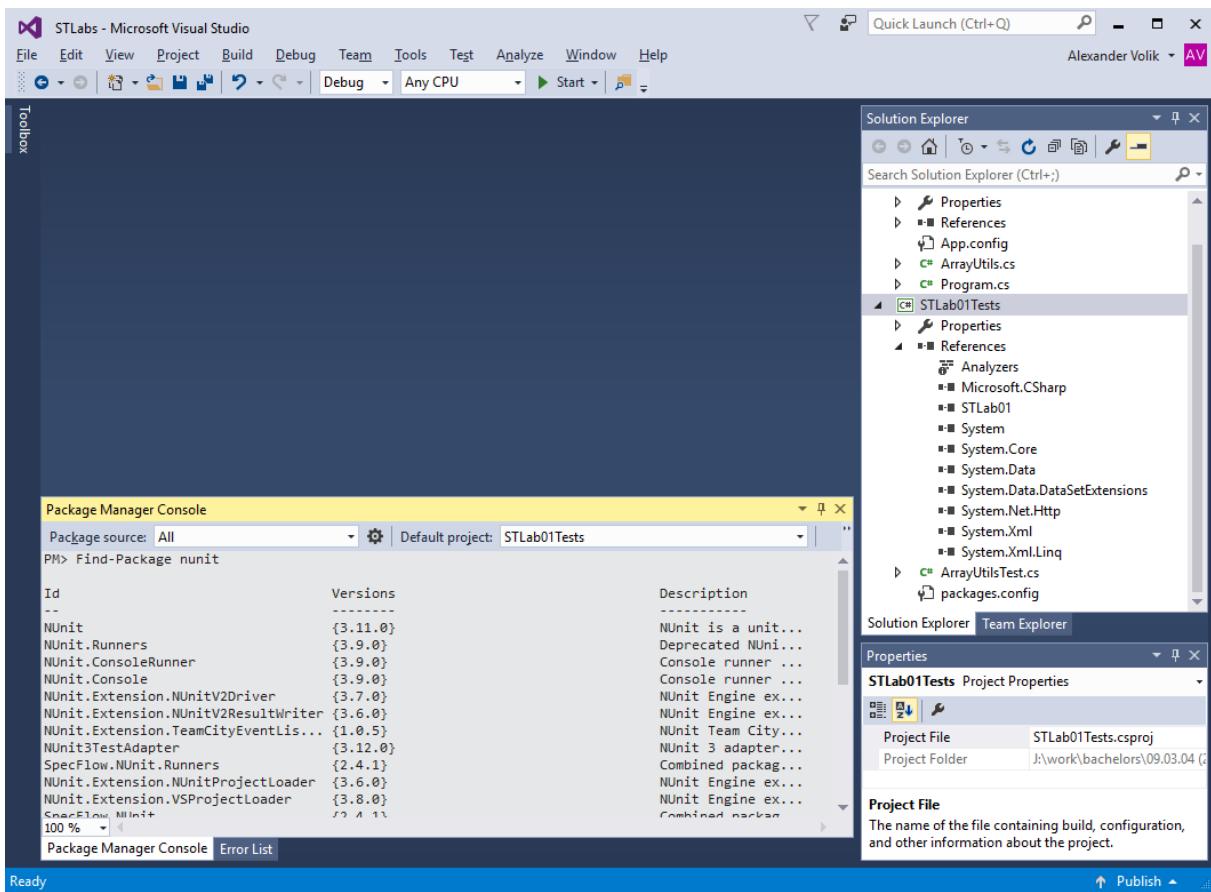


Рисунок С.9 – Поиск пакета в Package Manager Console

Добавим найденную библиотеку в тестовый проект, не забыв его выбрать в списке Default project. Для этого выполним команду "Install-Package NUnit". Результат её выполнения будет аналогичным использованию диалога "Add Library Package Reference" (рисунок С.10). Аналогичным образом можно установить библиотеку для любого проекта.

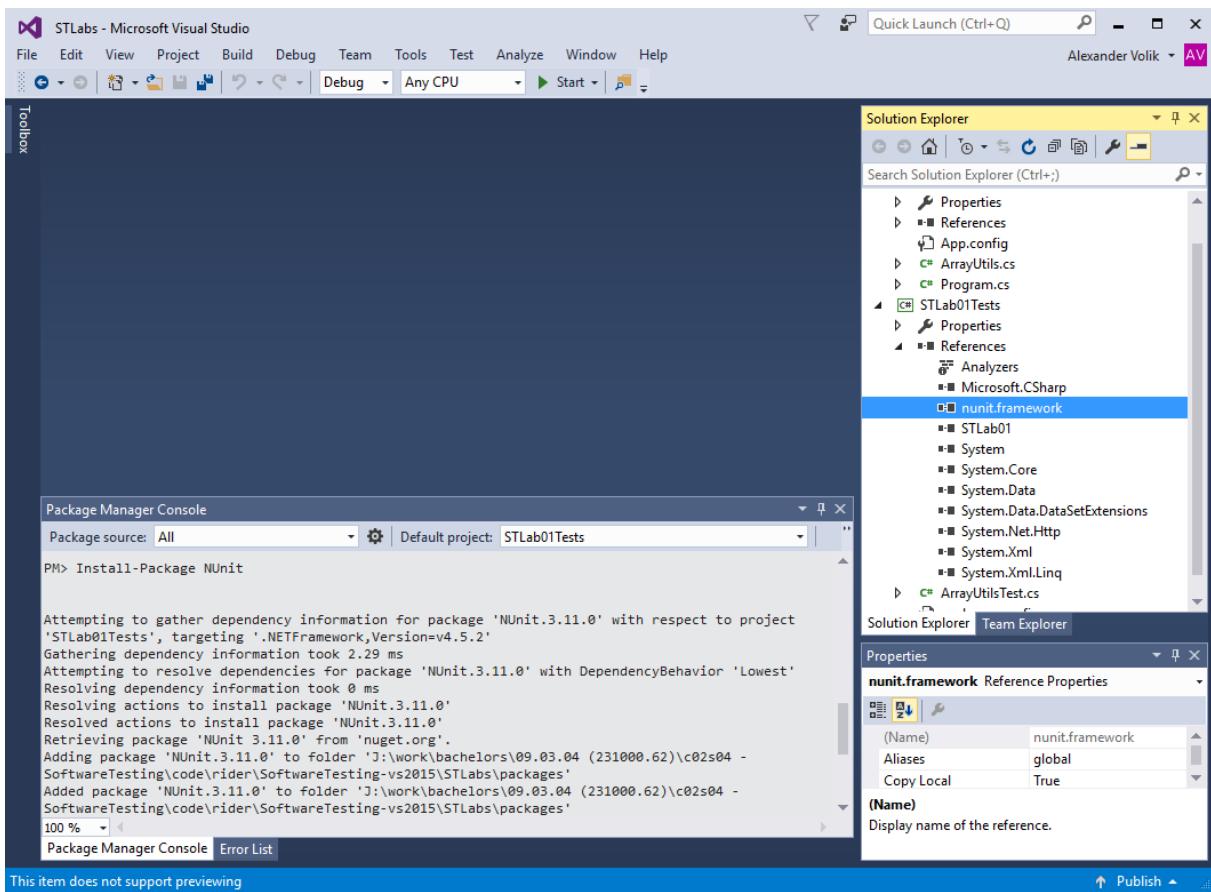


Рисунок С.10 – Установка пакета NUnit через Package Manager Console

Для удаления установленной библиотеки используется команда "Uninstall-Package".

В.3.3 Использования консоли для операций над группой проектов (конвейер команд)

Необходимо отметить еще одну возможность, которая предоставляет "Package Manager Console". Это возможность объединения команд в "конвейер" (pipeline). В частности это позволяет выбрать проект, к которому будет применена следующая команда. Для этого в начале строки необходимо указать:

- 1) Get-Project –All – чтобы выбрать все проекты текущего решения (solution).
- 2) Get-Project –Name *projectName* - чтобы выбрать проект с именем *projectName*.

После чего через разделитель "|" добавить необходимую команду. При этом нужно учесть, что не все поддерживают такой режим. Наиболее интересными представляются следующие варианты для установки,

обновления и удаления библиотеки, присутствующей во всех проектах решения:

```
Установка: Get-Project -All | Install-Package [params]
Обновление: Get-Project -All | Update-Package [params]
Удаление: Get-Project -All | Uninstall-Package [params]
```

B.3.4 Установка пактов из локального источника

Package Manager Console также позволяет выполнять установку пакетов без доступа к фиду из локального источника. Для этого необходимо указать параметр `-Source` с указанием пути к каталогу с пакетами. Например, если каталог с пакетами находится на диске F в папке packages, то команда будет выглядеть следующим образом:

```
Install-Package -Id NUnit -Source F:\\packages
```

B.5 Установка пактов с использованием dotnet CLI (dotnet core)

После выхода кроссплатформенной версии фреймворка .Net Core, в его составе появилась новая утилита для управления пакетами.

Она представляет собой интерфейс командной строки (command-line interface (CLI)) для предоставления удобных средств разработки приложений .NET.

Установка пакета через dotnet CLI выглядит следующим образом:

```
dotnet add package NUnit
```

После выполнения этой команды в файле проекта `.csproj` появится ссылка на добавленный пакет

```
<ItemGroup>
<PackageReference Include=" NUnit" Version="3.11.0" />
</ItemGroup>
```

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Лабораторный практикум

Составитель Волик Александр Георгиевич

Авторская правка

Редактор
Компьютерная верстка

А. Г. Волик

Подписано в печать
Бумага офсетная
Печ. л. 1,5
Усл. печ. л. 1,4
Уч.-изд. л. 1,0

Формат 60x84/16
Офсетная печать
Изд. №
Тираж 25 экз.
Заказ №

Цена руб.

Кубанский государственный технологический университет
350072, г. Краснодар, ул. Московская, 2, кор. А
Типография КубГТУ: 350058, г. Краснодар,
ул. Старокубанская, 88/4