Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования «Уральский федеральный университет имени первого Президента России Б.Н. Ельцина» (УрФУ)
Институт экономики и управления
Кафедра Анализа систем и принятия решений

Оценка _____
Руководитель междисциплинарного проектирования Чернильцев А.Г.
Члены комиссии _____
_____
Дата защиты _____

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к междисциплинарному проекту

Основы объектно-ориентированного программирования

Студент: Корнев Дмитрий Евгеньевич _____ _____
(ФИО) (Подпись)
Группа: ЭУ-193631

Екатеринбург
2020

Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования «Уральский федеральный университет имени первого Президента России Б.Н. Ельцина» (УрФУ)
Институт экономики и управления
Кафедра Анализа систем и принятия решений

Задание
на междисциплинарный проект

Студент Корнев Дмитрий Евгеньевич
группа ЭУ-193631
специальность/направление подготовки 38.03.05/05.01 Бизнес-информатика

1. Тема междисциплинарного проекта
Основы объектно-ориентированного программирования.

2. Содержание проекта, в том числе состав графических работ и расчетов
Введение
1. Обзор понятий и методов объектно-ориентированного анализа. Основы использования языка UML
2. Применение объектно-ориентированного подхода при программировании методов приближенных вычислений
3. Объектно-ориентированная разработка приложения для работы с одномерным массивом
Заключение
Список использованных источников

3. Дополнительные сведения
Оформлять пояснительную записку в соответствии с требованиями положения «Положение о выпускной
квалификационной работе (уровень бакалавриата) по направлениям подготовки»

4. План выполнения проекта:

Наименование элементов проектной работы	Сроки	Примечания	Отметка о выполнении
Выполнение 1 раздела	11.05.2020		
Выполнение 2 раздела	29.05.2020		
Выполнение 3 раздела	19.06.2020		
Заключение	25.06.2020		
Список использованных источников	25.06.2020		
Оформление	25.06.2020		
Нормоконтроль	26.06.2020		
Защита	30.06.2020		

Руководитель_____/ А. Г. Чернильцев /

Оглавление

Введение

Данная работа – итоговая картина усвоенных мною знаний за 1 курса по дисциплинам «Программирование» и «Объектно-ориентированный анализ и программирование». В ходе её подготовки и выполнения привлекались знания, полученные в ходе лекций и лабораторных работ.

Проект по модулю состоит из трёх частей. В первую входит раздел «Обзор понятий и методов объектно-ориентированного анализа. Основы использования языка UML» освещающий теоретический материал, вторая и третья посвящены практическому применению навыков и включают в себя разделы «Применение объектно-ориентированного подхода при программировании методов приближенных вычислений» и «Объектно-ориентированная разработка приложения для работы с одномерным массивом» соответственно.

В соответствии с индивидуальным заданием по варианту, в проекте мной будут рассмотрены технологии структурного и объектно-ориентированного программирования. В практической части работы будут продемонстрированы мои навыки разработки программ на платформе .NET Framework на языке C# (как консольное приложение, так и Windows Forms).

1. Обзор понятий и методов объектно-ориентированного анализа. Основы использования языка UML.

Тема: Сравнение технологий структурного и объектно-ориентированного программирования

Как и на любую сферу жизни, любое занятие, ремесло, на программирование можно взглянуть по-разному – использовать определённый подход к написанию программ, руководствоваться строгим набором принципов или же заимствовать их понемногу, применять подходящие к ситуации технологии: иными словами, следовать некоторому стилю разработки программ, написания кода.

Такие совокупности понятий и способов организации программного кода называются парадигмами программирования. Чаще всего выделяют две ключевых парадигмы: императивная и декларативная. Они, в свою очередь, подразделяются на многие другие, но в этом разделе проекта по модулю речь пойдёт о двух подразделах императивного программирования – структурном и объектно-ориентированном. Будут описаны и сравнены их основные принципы, показано применение в различных языках программирования. Важнейшие аспекты этих технологий будут визуализированы с помощью UML-диаграммы.

Перед описанием каждой из технологий отмечу, что нет универсального подхода к программированию, который был бы одинаково подходящим для любой задачи. Иногда разработчику приходится определять, какая технология легче найдёт место в создаваемом проекте, но зачастую преимущества структурного и объектно-ориентированного программирования комбинируются и используются вместе.

1.1. Принципы структурного программирования

Структурное программирование появилось как технология, предлагающая средства систематизации и рационализации создания и поддержки программного обеспечения, позволяющая повысить производительность труда разработчиков ПО, упростить процесс отладки и устранить проблемы, связанные со сложностью и практически полным отсутствием четкой иерархии в программном коде.

```
10 i = 0
20 i = i + 1
30 if i <= 10 then goto 70
40 if i > 10 then goto 50
50 print "Программа завершена."
60 end
70 print i; " в квадрате = "; i * i
80 goto 20
```

Рисунок 1 - Пример запутанного кода с несколькими переходами goto на языке Бейсик

Основоположником структурного программирования считается Эдсгер Дейкстра. В своей статье «Доводы против оператора GOTO» он отразил точку зрения, что качество программы существенно снижается от чрезмерного использования безусловных переходов. Данная концепция была твёрдо укреплена в программистском сообществе в начале 1970-х годов, когда в связи с развитием компьютерных технологий программы становились всё сложнее, и их было тяжелее понимать и сопровождать. Необходимо было решение, предполагающее разделение большой программы на блоки, подчиняющиеся друг другу, для их модульного использования, что позволило бы как оптимизировать программу, так и повысить читаемость кода, тем самым упростив его поддержку.

Главная идея структурного программирования в том, что абсолютно любая программа с использованием операторов `goto` может быть построена без него с помощью трёх конструкций – последовательности, ветвления и цикла, при этом для упрощения и повышения «человекочитаемости» программы код может быть разделён на подпрограммы или модули, которые при необходимости вызываются или подключаются соответственно.

Таким образом, можно выделить несколько основных принципов структурного подхода к программированию:

- Модульность

В программе имеются базовые управляющие конструкции: последовательность, ветвление и цикл. В разных языках программирования они представлены примерно

одинаковыми ключевыми словами или символами. Например, последовательное выполнение подпрограмм обозначается точкой с запятой (C++, C#), переносом строки (Python, Swift), ключевым словом then (в некоторых ситуациях -- Pascal/Delphi); ветвление обычно представлено блоками операторов `if-else` с незначительными вариациями, циклы описываются операторами `while`, `do..while`, `for` и пр.

Подкючение необходимых модулей/библиотек производится в начале описания кода программы по мере необходимости. Такой подход к хранению и использованию кода позволяет не задействовать лишние ресурсы рабочей машины и рассредоточить огромный массив кода из главной точки входа по разным файлам/каталогам. Более того, повышается его «реюзабилити», то есть, один и тот же модуль можно подключать и использовать в разных ситуациях, достигая абстракции (о ней позже).

Пример простой программы, построенной на принципе модульности, написанной на C (использованы все «принципиальные» конструкции, часть функционала выделена в отдельную подпрограмму):

```c
#include <stdio.h>

int number;

void init_number() {
    number = 8;
}

int main() {
    init_number();

    while (number < 41) {
        if (number % 5 == 0) {
            printf("%d", number);
        }
    }

    return 0;
}
```

- Подчинённость

Между подпрограммами и модулями связь организована «сверху вниз» – простыми словами, нельзя обратиться к чему-либо до его объявления. Если необходимо вызвать функцию или подпрограмму, она должна быть описана до вызова – иначе компилятор/интерпретатор попросту находится вне зоны видимости того, к чему

обращается программа. Это сделано для того, чтобы предотвратить ошибки, которые могут возникнуть от вызовов «в никуда» и спровоцировать серьёзные программные сбои.

В примере программы выше функция `init_number()` сначала объявлена, а потом вызвана в главной функции `main()`. Если описать эту функцию после main(), программа не скомпилируется, т.к. имя `init_number()` будет неизвестным.

- Локальность

Рассмотрим ещё один пример, на этот раз на языке C#:

```
using System;

namespace prog
{
    class Program
    {
        int globalNumber = 10;

        static void NumberIsHere()
        {
            int foreignNumber = 8;
        }

        static void Main()
        {
            int localNumber = 8;

            Console.WriteLine("{0}", globalNumber); // 10
            Console.WriteLine("{0}", localNumber);  // 9
            Console.WriteLine("{0}", foreignNumber);// Ошибка
        }
    }
}
```

Не будем обращать внимания на «объекто-ориентированные» ключевые слова `namespace` и `class`, посмотрим на функцию `Main()`.

В данном примере первые два вывода в консоль произойдёт успешно, а на третьем компилятор выдаст ошибку, так как переменная, которая указана в качестве параметра, ему недоступна. Имя `globalNumber` объявлено в общем пространстве, `localNumber` – в пределах функции `Main()`, а `foreignNumber` – вообще в другой функции, в своём локальном пространстве, вне которого эта переменная не видна. Такое разграничение доступа полезно для программиста, желающего абстрагироваться от кучи внешних имён и

сконцентрироваться на отдельной функции, не тратя время на отслеживание каждой переменной.

- Абстракция

Об этом принципе упоминалось в первом примере – пригодность модуля для его использования в разных программах. То есть, код подпрограммы пишется независимо от того, где он используется, и благодаря этому одна и та же функция может быть встроена в совершенно разные программные комплексы. Например, логическая функция, определяющая наложение двух прямоугольников друг на друга, может быть использована как при разработке компьютерной игры для распознавания столкновений объектов, так и в приложении для визуализации геометрических фигур – при наложении двух фигур друг на друга у их общей площади меняется прозрачность или выдается какое-то сообщение. Всё зависит от задач, поставленных перед разработчиком. В первом примере программы в качестве модуля используется стандартная библиотека ввода-вывода `stdio`, заголовочный файл которой подключается в начале программного кода.

Визуализировать структурный подход можно с помощью следующей UML-диаграммы на примере покупки билетов в приложении автовокзала:

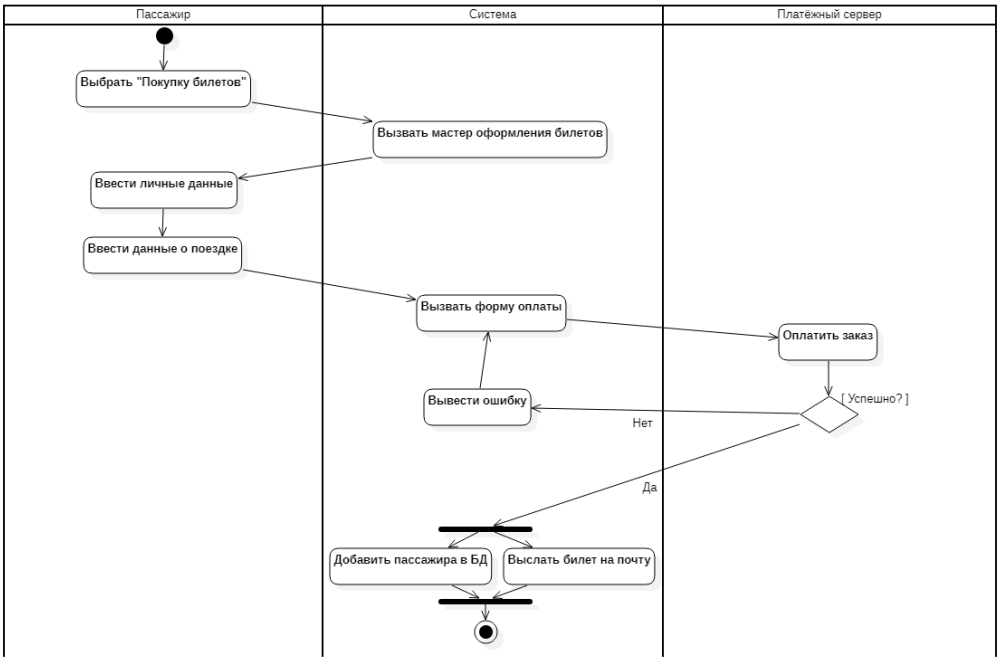

Рисунок 2 - UML-диаграмма процесса покупки билета в приложении автовокзала, демонстрирующая ряд принципов структурного программирования

Уже здесь можно увидеть сразу два реализованных принципа – модульности и последовательности. Реализация остальных аспектов остаётся за программистом. Действия выполняются друг за другом, используются операторы ветвления и цикла (при

проверке ошибки), а сам процесс покупки билетов разделён между тремя модулями, к которым программа обращается по мере необходимости. В UML принципы структурного программирования просматриваются во многих элементах – прежде всего, в элементе Control Flow, передающего дальнейший контроль следующему элементу, в ветвлении Decision, поведенческих диаграммах взаимодействий Interaction, последовательностей Sequence и других.

Важно отметить, что на диаграмме нет отдельно стоящих элементов – к каждому ведёт связь, а это позволяет сделать вывод, что в структурном подходе к созданию программ практически нет избыточных операций, и используется весь программный код.

1.2. Принципы объектно-ориентированного программирования

Объектно-ориентированный подход к разработке программного предполагает представление программы как упорядоченного набора объектов классов. В целом объектно-ориентированное программирование реализует основную конечную цель структурного программирования – повышение понимания структуры программы и качества кода, но в более полной мере – с помощью понятий классов, их экземпляров, наследования, абстракции данных, инкапсуляции и полиморфизма.

Объектно-ориентированное программирование нашло применение в крупных проектах за его детальный и абстрактный подход к работе с данными, получаемыми от пользователя. Это позволяет наиболее точно смоделировать бизнес-логику приложения, тем самым повышая его эффективность для конечной аудитории.

ООП имеет свою терминологию. Переменные, объявленные внутри классов и являющиеся параметрами его объектов, называются полями класса, процедуры и функции, принадлежащие классу – методами. Инкапсулируются элементы класса с помощью установки уровня и методов доступа. Полиморфизм достигается за счёт перегрузок методов. Некоторые поля класса называются свойствами, чаще в тех ситуациях, когда права на чтение и запись различны по уровню доступа.

Рассмотрим принципы объектно-ориентированного подхода подробнее.

- Абстракция данных

В программном коде отбрасывается не значимая в данный момент информация, а основной фокус берётся на необходимые данные для реализации конкретной задачи. Рассмотрим следующий код:

```
static class Stuff
{
    static void PrintSomething()
    {
        Console.WriteLine("Hey there!");
    }
}

class Program
{
    static void Main()
    {
        Stuff.PrintSomething()

        Console.ReadKey();
    }
}
```

Данный код не совсем правильный с точки зрения ООП. Эта программа заточена под выполнение исключительно в «консольной» среде, то есть строка «Hey there!» может быть выведена исключительно в ней, но не в каком-либо элементе формы, например. Это сильно ограничивает область применения класса `Stuff`. Гораздо рациональнее было бы реализовать метод, возвращающий строку «Hey There!» и вызвать его в главной точке входа `Main()` как параметр в методе `Console.WriteLine()`:

```
static class Stuff
{
    static string PrintSomething()
    {
        return "Hey there!";
    }
}

class Program
{
    static void Main()
    {
        Console.WriteLine(Stuff.PrintSomething());

        Console.ReadKey();
    }
}
```

В этом примере гораздо цельнее соблюдена абстракция данных и «обязанности» каждого класса не перекладываются друг на друга.

- Инкапсуляция

Инкапсуляцией называется размещение в классе методов и полей с сокрытием их внутренней реализации. То есть, «снаружи» доступен только вызов метода или, например,

чтение (без записи) свойства. В следующем примере свойству Name выставлено разрешение на чтение, но запись может производиться только внутри класса:

```
class Bot
{
    public string Name { get; private set; }

    public Bot()
    {
        Name = "Bronislav";
    }
}

class Program
{
    static void Main()
    {
        var bot = new Bot();

        Console.WriteLine(bot.Name);    // Bronislav
        bot.Name = "Albert";            // Ошибка!
        Console.ReadKey();
    }
}
```

Ошибку можно устранить изменением уровня доступа к записи на `public`, но тогда нарушится инкапсуляция данных. С точки зрения ООП, лучшим решением будет написать метод, задающий новое значение, переданное в качестве входного параметра:

```
class Bot
{
    public string Name { get; private set; }

    public Bot()
    {
        Name = "Bronislav";
    }

    public void NewName(string newName)
    {
        Name = newName;
    }
}

class Program
{
    static void Main()
    {
        var bot = new Bot();

        Console.WriteLine(bot.Name);    // Bronislav
        bot.NewName("Albert");
        Console.WriteLine(bot.Name);    // Albert
```

```
            Console.ReadKey();
        }
    }
```

- Полиморфизм

Это понятие означает способность одной функции (в случае ООП – метода) обрабатывать разные типы данных. При разных наборах аргументов с разными типами реализация функции меняется. Так, работа метода с одним и тем же именем может быть определена одним способом для входного параметра строкового типа и другим способом для целочисленного параметра. Полиморфизм реализуется с помощью перегрузок методов. Например, в стандартной библиотеке `System` языка C# присутствует масса перегруженных методов, которые могут работать с разными наборами аргументов.

- Наследование

Позволяет при уже описанном классе с полями, свойствами и методами не копировать содержимое в новый класс для модификации/переопределения/расширения функционала, а частично или полностью перенести его в дочерний класс (класс-потомок). Наследование в основном используется для исключения неоправданных повторений в коде и для выделения общих черт у объектов разных дочерних классов в общий родительский класс.

С помощью UML-диаграммы можно также визуализировать принципы ООП:

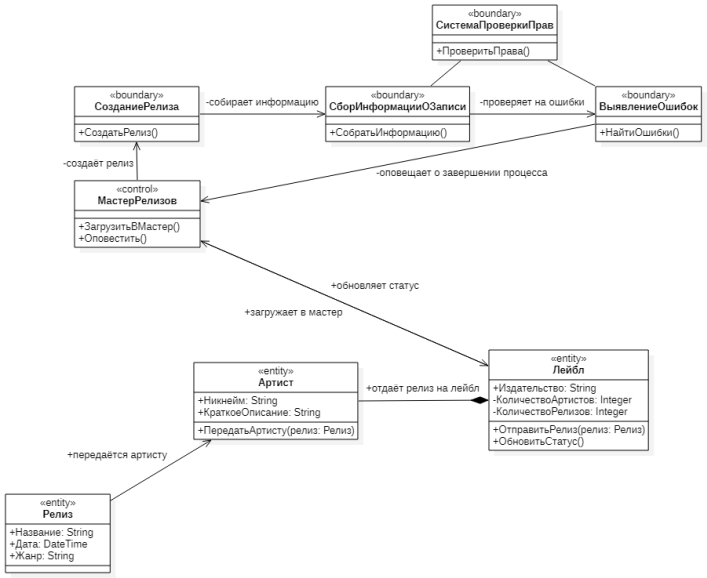

Рисунок 3 - UML-диаграмма, отражающая связь разных классов на примере центра управления музыкальными релизами

UML предоставляет возможность реализовать принципы ООП в полной мере: на примере выше описаны классы с соблюдением инкапсуляцию данных в виде модификаторов доступа (на диаграмме обозначены знаками public «+» и private «-», также существуют обозначения для protected «#» и др.). Среди средств UML присутствуют элементы ассоциации, агрегации и наследования (Association, Aggregation, Generalization), позволяющие построить полноценную диаграмму с объектно-ориентированным подходом.

1.3. Сравнение технологий

Структурный подход к анализу и программированию предполагает, что разработчик разбивает программу на уровни абстракции с ограниченным числом элементов и создаёт программный код «сверху вниз». Это подталкивает к использованию так называемых «заглушек» – функций, которые не выполняют никакой работы и лишь символизируют определённый программный интерфейс, а их функционал наполняется по мере разработки программы. В то же время при объектно-ориентированном подходе, изначально сосредоточенном на моделировании рабочей логики программы, разработчик описывает все необходимые классы, их поля и методы, и лишь потом задаёт вызовы в главной точке входа.

В отличие от ООП, структурное программирование не предполагает классов и объектов вообще – вместо этого используются глобальные и локальные переменные, а

уровень доступа определяется в зависимости от контекста. В ряде ситуаций, например, работе с несколькими программными модулями, это затрудняет взаимодействие имён, хранящихся в разных пространствах. В объектно-ориентированном программировании такая проблема решается с помощью регулирования доступа к данным модификаторами `public`, `protected`, `private` и др. К сожалению, технология ООП неизбежно затрачивает больше вычислительных ресурсов на выполнение программы, чем структурный подход, так как компилятору/интерпретатору необходимо постоянно определять, какие методы будет вызываться в течение выполнения программы.

Общей чертой этих двух технологий можно назвать направленность на повторное использование кода, повышение удобства создания программного обеспечения, его поддержки, доработки, отладки и расширения функционала. Несомненно, появление концепции структурного программирования и её развитие сильно повлияло на дальнейшее развитие информационных технологий, в том числе и на появление ООП. Объектно-ориентированный подход позволил разработчикам ПО концентрироваться на определённой задаче, а не на всей программе сразу, разделять обязанности в команде и создавать более эффективные продукты, заточенные под конечного пользователя.

2. Применение объектно-ориентированного подхода при программировании методов приближенных вычислений

2.1. Цель работы

Необходимо создать приложение на языке C#, в котором выполняется приближенное вычисление интеграла определенной функции разными методами численного интегрирования (методы прямоугольников, метод трапеций и метод Симпсона), выводятся графики функций с указанием интервала интегрирования.

Из предоставленного выбора между графическим и консольным интерфейсами приложения для своей работы я выбрал консольный, т.к. на мой взгляд, он более универсальный для использования в различных операционных системах, например, не только в Windows, но в Unix-подобных.

По моему варианту нужно работать со следующими двумя функциями:

$$f_1 = \frac{8}{5 + 2^{x^2}}$$

$$f_2 = \frac{x^2}{(x+8)^{\frac{1}{3}}}$$

Вышеописанные функции должны быть проинтегрированы тремя способами: методом средних прямоугольников, методом трапеций и методом Симпсона «3/8». Также функция интегрируется программой аналитическим способом, то есть, заранее заложенной в ней формулой уже найденной первообразной.

Структура программы включает в себя три модуля:

- Модуль классов, содержащий инструментарий для интегрирования функций;
- Модуль, в котором описаны сами интегрируемые функции;
- «Связующий» модуль, предоставляющий пользовательский интерфейс командной строки и взаимодействие с модулями выше.

2.2. Разработка приложения

Приложение разрабатывалось на языке C#. Для написания кода программы использовался интерактивный текстовый редактор Visual Studio Code, а для отладки и тестирования было создано решение function-integration в среде разработки MS Visual Studio 2019 на основе шаблона «Консольное приложение (.NET Framework)», куда были добавлены уже написанные модули.

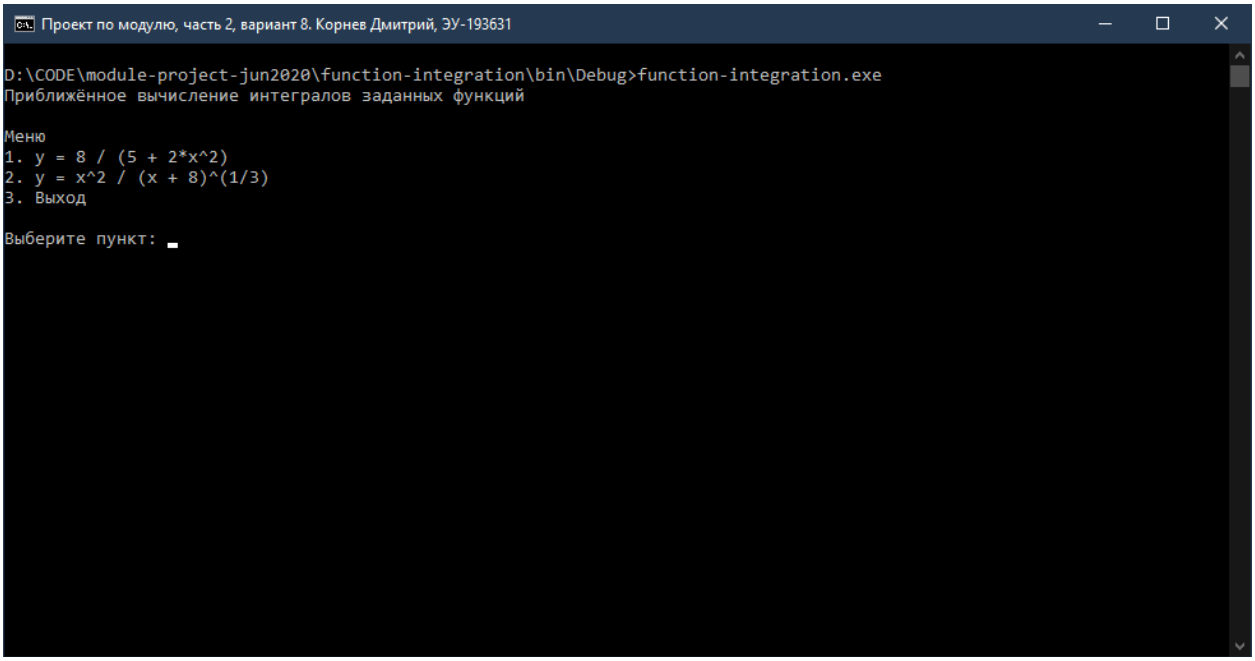

Рисунок 4 - стартовое меню приложения

Модуль IntegrationMethods.cs

Основополагающий модуль программы, содержащий в себе абстрактный класс `IntegrationMethods`. Он содержит в себе объявления трёх абстрактных методов, которые при наследовании необходимо переопределить. Они представляют интегрируемую функцию с аргументом *x*, её аналитически вычисленную первообразную с аргументами границ интегрирования и удобочитаемое написание (для вывода в консоль):

```
abstract public double Y(double x);
abstract public double IntY(double low, double hi);
abstract public string FuncSpelling();
```

Также класс содержит наследуемые методы `MidRectMethod`, `TrapeziaMethod` и `Simpson2Mehod`, которые вычисляют значение интеграла методом средних треугольников, трапеции и Симпсона «3/8» соответственно, и возвращают его. Для расчёта значений аналитически использовались формулы с сайта https://www.cleverstudents.ru.

Эти методы представлены ниже:

```
public double MidRectMethod(double low, double intStep, double stepCount)
{ // интегрирование методом трапеций
    double result = 0, x = low;

    for (int i = 1; i < stepCount; i++)
    {
        result += Y(x - (intStep / 2)) * intStep;
        x += intStep;
    }

    return result;
}

public double TrapeziaMethod(double low, double intStep, double stepCount)
{ // интегрирование методом трапеций
    double result = 0, x = low;

    for (int i = 1; i < stepCount; i++)
    {
        result += (Y(x) + Y(x - intStep)) / 2 * intStep;
        x += intStep;
    }

    return result;
}

public double Simspon2Method(double low, double intStep, double stepCount)
{ // метод для интегрирования методом Симпсона 3/8
    double result = 0, x = low;

    for (int i = 1; i < stepCount; i++)
    {
        /*
            для удобства числитель разбит на
            три дополнительные переменные
        */
        double a = Y(x - intStep);
        double b = 3 * Y((2 * (x - intStep) + x) / 3);
        double c = 3 * Y((x - intStep + 2 * x) / 3);

        result += (a + b + c + Y(x)) / 8 * intStep;
        x += intStep;
    }

    return result;
}
```

Модуль FunctionData.cs

Модуль, хранящий в себе описания интегрируемых функций по варианту, их аналитически вычисленные первообразные и строковое представление.

Сами функции хранятся в статическом классе `Funcs` в виде методов `Function1` и `Function2`:

```
public static class Funcs
{
    public static double Function1(double x)
    {
        return 8 / (5 + 2 * x * x);
    }

    public static double Function2(double x)
    {
        return (x * x) / Math.Pow(x + 8, (double)1 / 3);
    }
}
```

Первообразные функций хранятся в классе `FuncInts` в таком же виде, только у методов в именах присутствует суффикс `Int`:

```
public static class FuncInts
{
    public static double FunctionInt1(double x)
    {
        return 8 * Math.Atan(Math.Sqrt(2) * x / Math.Sqrt(5)) / Math.Sqrt(10);
    }

    public static double FunctionInt2(double x)
    {
        return (Math.Pow(x + 8, (double)2 / 3) *
                (15 * x * x - 144 * x + 1728)) / 40;
    }
}
```

Чтобы функции можно было вызывать в пользовательском интерфейсе, они должны унаследовать класс IntegrationMethods и переопределить его абстрактные методы под возвращение соответствующих значений. Для этого я объявил для каждой функции дочерние классы Function1 и Function2 соответственно и в каждом из них переопределил методы `Y()`, `IntY()` и `FuncSpelling()` с помощью ключевого слова `override`.

```
public class Function1 : IntegrationMethods
{
    public override double Y(double x)
    {
        return Funcs.Function1(x);
    }

    public override double IntY(double low, double hi)
    {
        return FuncInts.FunctionInt1(hi) - FuncInts.FunctionInt1(low);
    }

    public override string FuncSpelling()
    {
        return "y = 8 / (5 + 2*x^2)";
    }
}

public class Function2 : IntegrationMethods
{
    public override double Y(double x)
    {
        return Funcs.Function2(x);
    }

    public override double IntY(double low, double hi)
    {
        return FuncInts.FunctionInt2(hi) - FuncInts.FunctionInt2(low);
    }

    public override string FuncSpelling()
    {
        return "y = x^2 / (x + 8)^(1/3)";
    }
}
```

Модуль Program.cs

Основной модуль, обеспечивающий связь вышеописанных операционных модулей с пользовательским интерфейсом командной строки. В сгенерированном по умолчанию классе `Program` объявлено статическое поле `function`, являющееся экземпляром абстрактного класса `IntegrationMethods`.

```
public static IntegrationMethods function;
```

Чаще всего объявляются экземпляры дочерних классов с уже уточнённым и переопределённым функционалом, но в данном случае использование «родительского» экземпляра оправдано – в начале работы программы возможен выбор интегрируемой

функции посредством меню, и неизвестно, с какой функцией решит работать пользователь. Поэтому этот универсальный экземпляр можно инициализировать любым из потомков его класса, что и будет сделано во время подготовки к вычислениям.

Всё действие происходит в методе `Main()` – в данном случае можно себе позволить такую нагрузку, т.к. весь функционал вынесен в отдельные модули, и программа лишь оперирует уже готовым алгоритмом действий.

В начале метода объявляются вещественные переменные для границ интегрирования, количества шагов и байтовая переменная для хранения выбранного пункта в меню (тип данных `byte` выбран для экономии памяти).

```
double lowerBound = 0, upperBound = 1, stepCount = 20;
byte choice = 0;
```

Далее задаётся заголовок окна консоли и выводится название работы:

```
Console.Title = "Проект по модулю, часть 2, вариант 8. Корнев Дмитрий, ЭУ-193631";
Console.WriteLine("Приближённое вычисление интегралов заданных функций");
```

После этого запускается главный цикл программы, выход из которого обеспечивается через меню – после чего программа закрывается. В начале цикла выводится непосредственно текстовое меню с вариантами выбора двух функций, либо выхода из приложения:

```
Console.Write("\nМеню\n" +
              "1. y = 8 / (5 + 2*x^2)\n" +
              "2. y = x^2 / (x + 8)^(1/3)\n" +
              "3. Выход\n\n");
```

Чтобы при неверно введённом значении (например, 8 или «а») программа не завершалась с ошибкой, введена логическая переменная `inputFail`, которая хранит ложное значение, пока не происходит верного ввода, и блок try-catch, проверяющий верность типа данных. В случае несоответствия типа блок отправляет в меню значение 0, что тоже является неверным, поэтому цикл начинается заново и пользователь повторяет ввод.

Если получено верное значение (от 1 до 3) выбирается один из пунктов меню с помощью блока switch-case и программа работы продолжается. Ниже представлен полностью код всего цикла проверки, т.к. разрывать его на части было не совсем правильно:

```
while (inputFail)
{
	Console.Write("Выберите пункт: ");
	try
	{
		choice = Convert.ToByte(Console.ReadLine());
	}
	catch
	{
		choice = 0;
	}

	switch (choice)
	{
		case 1:
			function = new Function1();
			inputFail = false;
			break;
		case 2:
			function = new Function2();
			inputFail = false;
			break;
		case 3:
			Console.ForegroundColor = ConsoleColor.Cyan;
			Console.WriteLine("До свидания!\n");
			Console.ResetColor();
			return;
		default:
			Console.ForegroundColor = ConsoleColor.Red;
			Console.WriteLine("Неверный выбор.\n");
			Console.ResetColor();
			break;
	}
}
```

Следующим шагом переменным `lowerBound`, `upperBound` и `stepCount` присваиваются значения нижней и верхней границ и количества шагов интегрирования соответственно. Проверка значений контролируется блоком try-catch в комбинации с циклом `while` во избежание необработанных исключений (состояние ввода контролирует всё та же переменная `inputFail`):

```
Console.Write("\nНижняя граница = ");
lowerBound = Convert.ToDouble(Console.ReadLine());
Console.Write("Верхняя граница = ");
upperBound = Convert.ToDouble(Console.ReadLine());
Console.Write("Количество шагов = ");
stepCount = Convert.ToDouble(Console.ReadLine());
```

Далее вычисляется шаг интегрирования в объявленной для его хранения переменной:

```
double intStep = (upperBound - lowerBound) / stepCount;
```

После того, как шаг интегрирования вычислен, можно посчитать интегралы разными методами, в том числе и аналитическим:

```
var midRectsResult = function.MidRectMethod(lowerBound, intStep, stepCount);
var trapeziaResult = function.TrapeziaMethod(lowerBound, intStep, stepCount);
var simpson2Result = function.Simspon2Method(lowerBound, intStep, stepCount);
var analyticResult = function.IntY(lowerBound, upperBound);
```

И когда всё, наконец, вычислено, можно выводить результат:

```
Console.WriteLine("Результаты численного интегрирования\n" +
"-----------------------------------------------------------------------\n" +
      $"Функция {choice}. {function.FuncSpelling()}; " +
      $"шаг {intStep:f2}; a={lowerBound}; b={upperBound}; s = {analyticResult:f8}");

Console.WriteLine($"Метод средних прямоугольников | {midRectsResult,11:f8}, " +
      $"погрешность: {Math.Abs(analyticResult - midRectsResult),-11:f8}\n" +
      $"Метод трапеций                 | {trapeziaResult,11:f8}, " +
      $"погрешность: {Math.Abs(analyticResult - trapeziaResult),-11:f8}\n" +
      $"Метод Симпсона 3/8             | {simpson2Result,11:f8}, " +
      $"погрешность: {Math.Abs(analyticResult - simpson2Result),-11:f8}");

Console.WriteLine(
"-----------------------------------------------------------------------");
```

После вывода результата программа возвращается в начало основного цикла и предлагает повторно выбрать функцию для вычислений с новыми значениями, либо выйти из программы.

2.3. Диаграмма классов

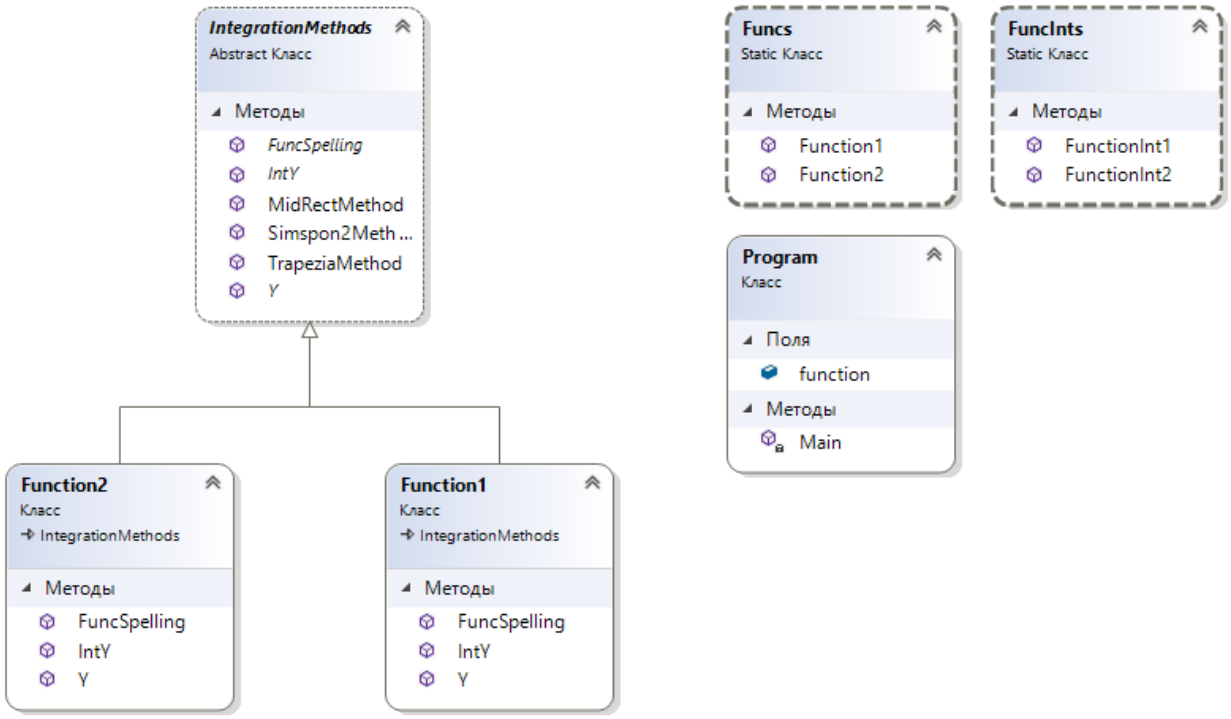

Рисунок 5 - Развёрнутая диаграмма классов проекта function-integration

2.4. Тестирование

При запуске программы в командную строку выведется меню с тремя предложенными вариантами. Если ввести вариант не из списка – будет вызвано исключение, но программа не завершится, а предложит повторить ввод:

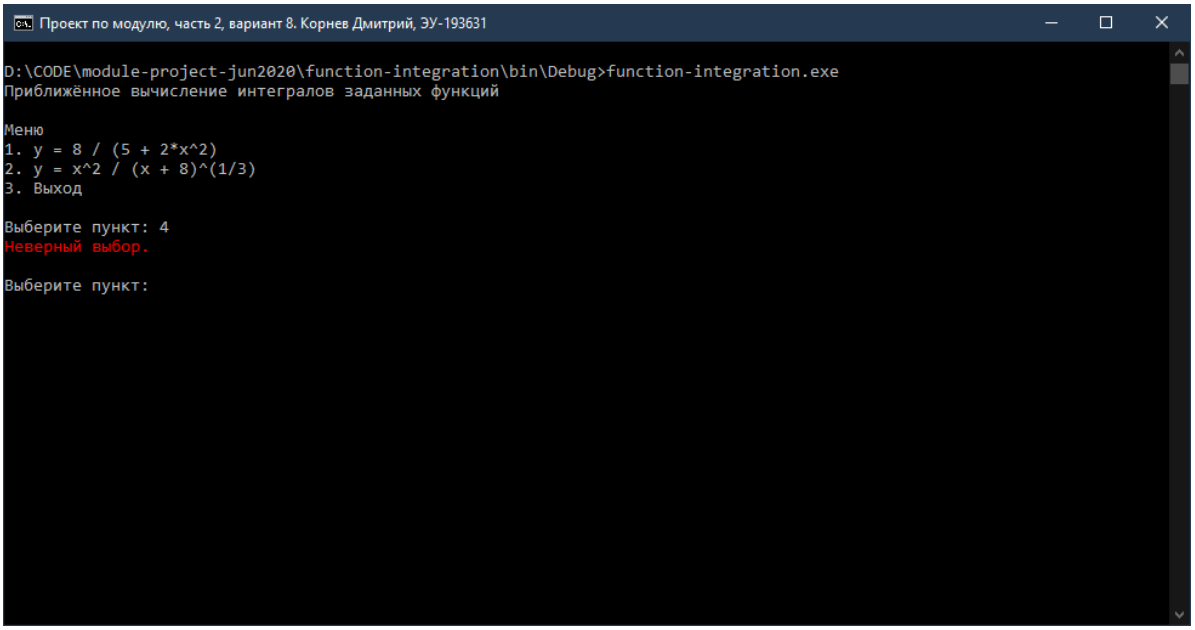

Рисунок 6 - обработанное исключение неверного выбора пункта меню

Если выбрать одну из двух функций и ошибиться во введённом значении, например, нижней границы, программа так же на это среагирует и предложит начать ввод параметров заново (но уже в пределах выбранной функции):

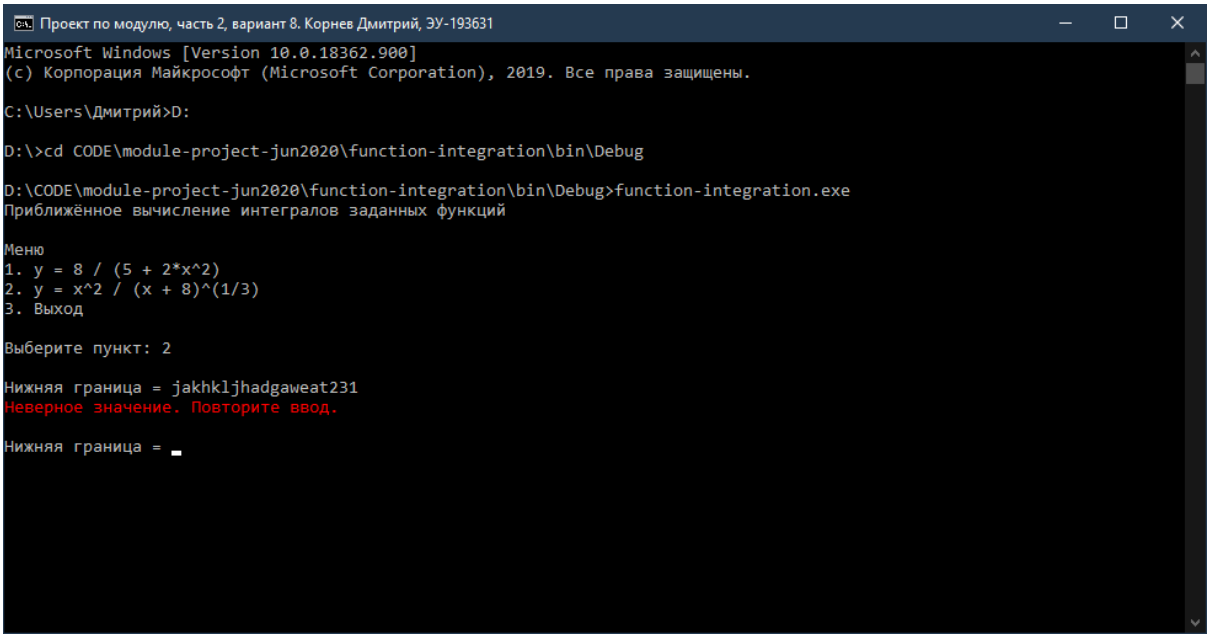

Рисунок 7 - Обработанное исключение неверного значения границы

Перейдём к вычислениям интегралов. Если выбрать малое количество шагов, например, 2 – погрешность будет огромной:

```
Выберите пункт: 1

Нижняя граница = 0
Верхняя граница = 1
Количество шагов = 2
Результаты численного интегрирования
-----------------------------------------------------------------
Функция 1. y = 8 / (5 + 2*x^2); шаг 0,50; a=0; b=1; s = 1,42667457
Метод средних прямоугольников |  0,78048780, погрешность: 0,64618677
Метод трапеций                |  0,76363636, погрешность: 0,66303821
Метод Симпсона 3/8            |  0,77484643, погрешность: 0,65182814
-----------------------------------------------------------------
```

Рисунок 8 - Пример вычислений для первой функции с двумя шагами

Если увеличить количество шагов, то погрешность, естественно, сокращается:

```
Выберите пункт: 1

Нижняя граница = 0
Верхняя граница = 1
Количество шагов = 200
Результаты численного интегрирования
-----------------------------------------------------------------
Функция 1. y = 8 / (5 + 2*x^2); шаг 0,01; a=0; b=1; s = 1,42667457
Метод средних прямоугольников |  1,42321399, погрешность: 0,00346058
Метод трапеций                |  1,42321193, погрешность: 0,00346264
Метод Симпсона 3/8            |  1,42321331, погрешность: 0,00346127
-----------------------------------------------------------------
```

Отмечу, что самым точным остаётся метод средних прямоугольников, метод Симпсона 3/8 занимает второе место, а дальше всех от идеала оказался метод трапеций.

График интеграла функции, построенный с помощью сервиса Geogebra:

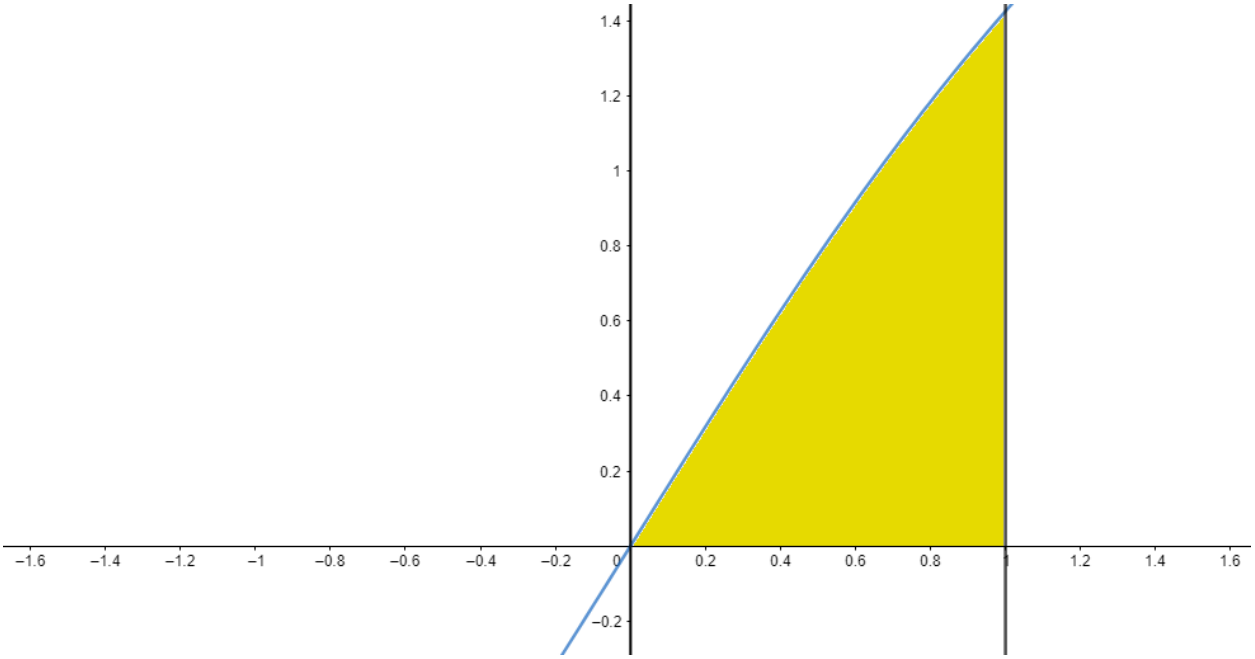

Рисунок 9 - Интеграл функции 1, интервал [0; 1]

Для второй функции повторю те же самые операции (кстати, округленное до 8 знаков после запятой аналитическое значение совпало со значением в таблице вариантов – 0,1617797191):

```
Нижняя граница = 0
Верхняя граница = 1
Количество шагов = 20
Результаты численного интегрирования
-----------------------------------------------------------------
Функция 2. y = x^2 / (x + 8)^(1/3); шаг 0,05; a=0; b=1; s = 0,16177972
Метод средних прямоугольников |  0,11820069, погрешность: 0,04357903
Метод трапеций                |  0,11848322, погрешность: 0,04329650
Метод Симпсона 3/8            |  0,11829487, погрешность: 0,04348485
-----------------------------------------------------------------
```

Рисунок 10 - Пример для второй функции, 20 шагов

```
Нижняя граница = 0
Верхняя граница = 1
Количество шагов = 2000
Результаты численного интегрирования
-----------------------------------------------------------------
Функция 2. y = x^2 / (x + 8)^(1/3); шаг 0,00; a=0; b=1; s = 0,16177972
Метод средних прямоугольников |  0,16129943, погрешность: 0,00048029
Метод трапеций                |  0,16129946, погрешность: 0,00048026
Метод Симпсона 3/8            |  0,16129944, погрешность: 0,00048028
-----------------------------------------------------------------
```

Рисунок 11 - Пример для второй функции, 2000 шагов

Вычисления выполнялись на отрезке [0, 1], и здесь лидером по точности оказался метод трапеций, метод Симпсона 3/8 остался на втором месте, а третье занял метод средних прямоугольников.

График интеграла функции, построенный с помощью сервиса Geogebra:

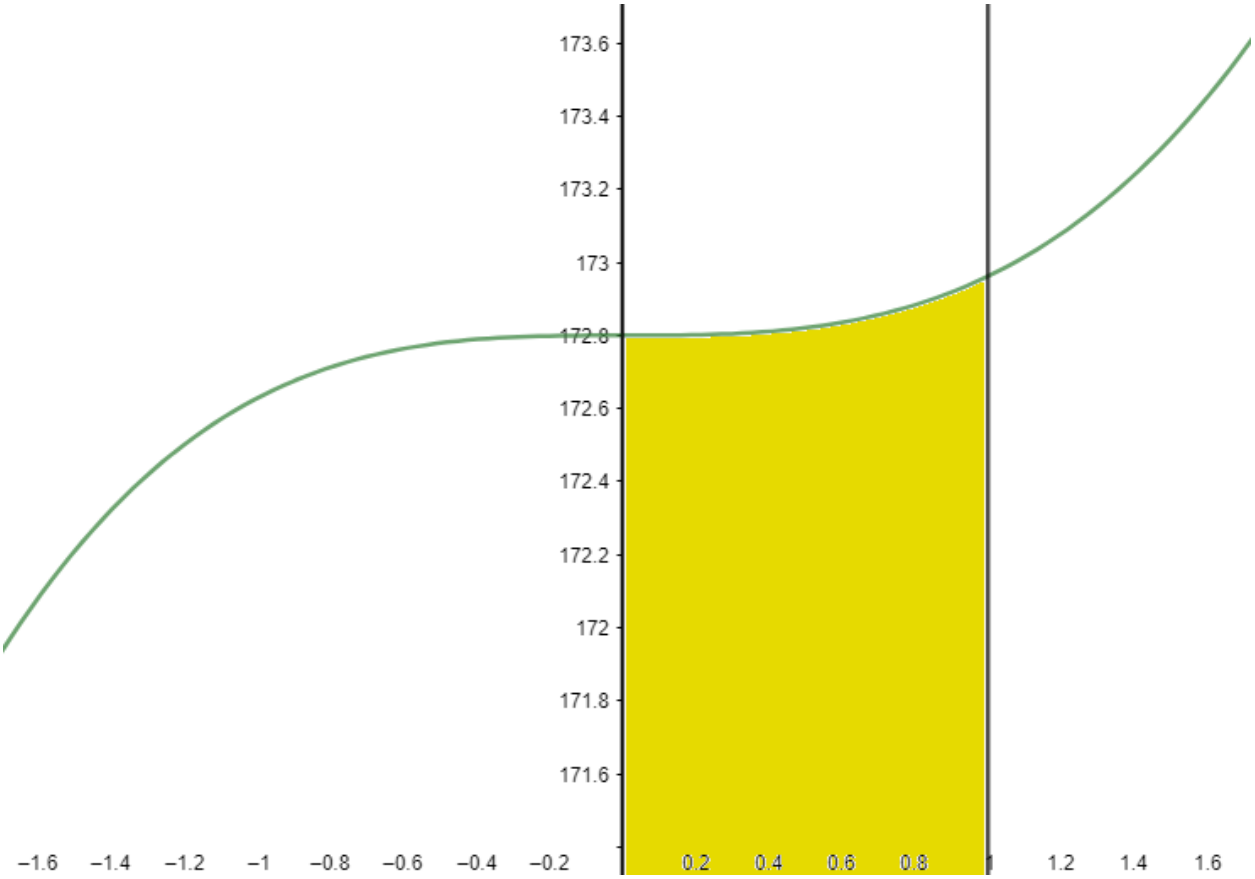

Рисунок 12 - Интеграл функции 2, интервал [0;1]

После того, как я протестировал обе функции, я вышел из программы с помощью пункта 3 в меню:

```
Меню
1. y = 8 / (5 + 2*x^2)
2. y = x^2 / (x + 8)^(1/3)
3. Выход

Выберите пункт: 3
До свидания!


D:\CODE\module-project-jun2020\function-integration\bin\Debug>
```

Рисунок 13 - Выход из программы

2.5. Вывод

В ходе разработки приложения я следовал парадигме объектно-ориентированного программирования. Я закрепил навыки работы с классами, абстракциями, наследованием и использовал различные базовые инструменты языка C#, такие как форматированный вывод и математические функции. Также на практике выяснил, какой из представленных в варианте работы методов вычисления интеграла эффективнее для конкретной функции.

3. Объектно-ориентированная разработка приложения для работы с одномерным массивом

3.1. Цель работы

В среде Microsoft Visual C# необходимо создать приложение, в котором создаются, наследуются и применяются классы и объекты при работе с одномерным числовым массивом, а также используются графические средства для визуализации числовых данных массива и операций, выполняемых с этими данными.

Код программы было решено разделить на три модуля вместо двух:

- ArrayKit.cs – модуль классов, обеспечивающих работу с целочисленным массивом;
- Form1.cs – модуль формы, предоставляющей графический интерфейс приложения и управляние компонентами программы;
- Histogram.cs – отдельный модуль для статического класса, отвечающего за построение гистограммы в форме.

По моему варианту с массивом необходимо выполнить следующие операции:

№	Задание для одномерного массива	Размещение чисел в файле	Способ сортировки
8.	Получить новый массив из сумм соседних элементов исходного массива	В одной строке через пробел	Selection2

3.2. Разработка приложения

Разработка программы велась на языке C#. Приложение создавалось с помощью нескольких средств разработки: код модуля классов разрабатывался в редакторе кода Visual Studio Code, графический интерфейс формы создавался в среде разработки Visual Studio 2019. Все файлы проекта были объединены в решение one-dim-array.

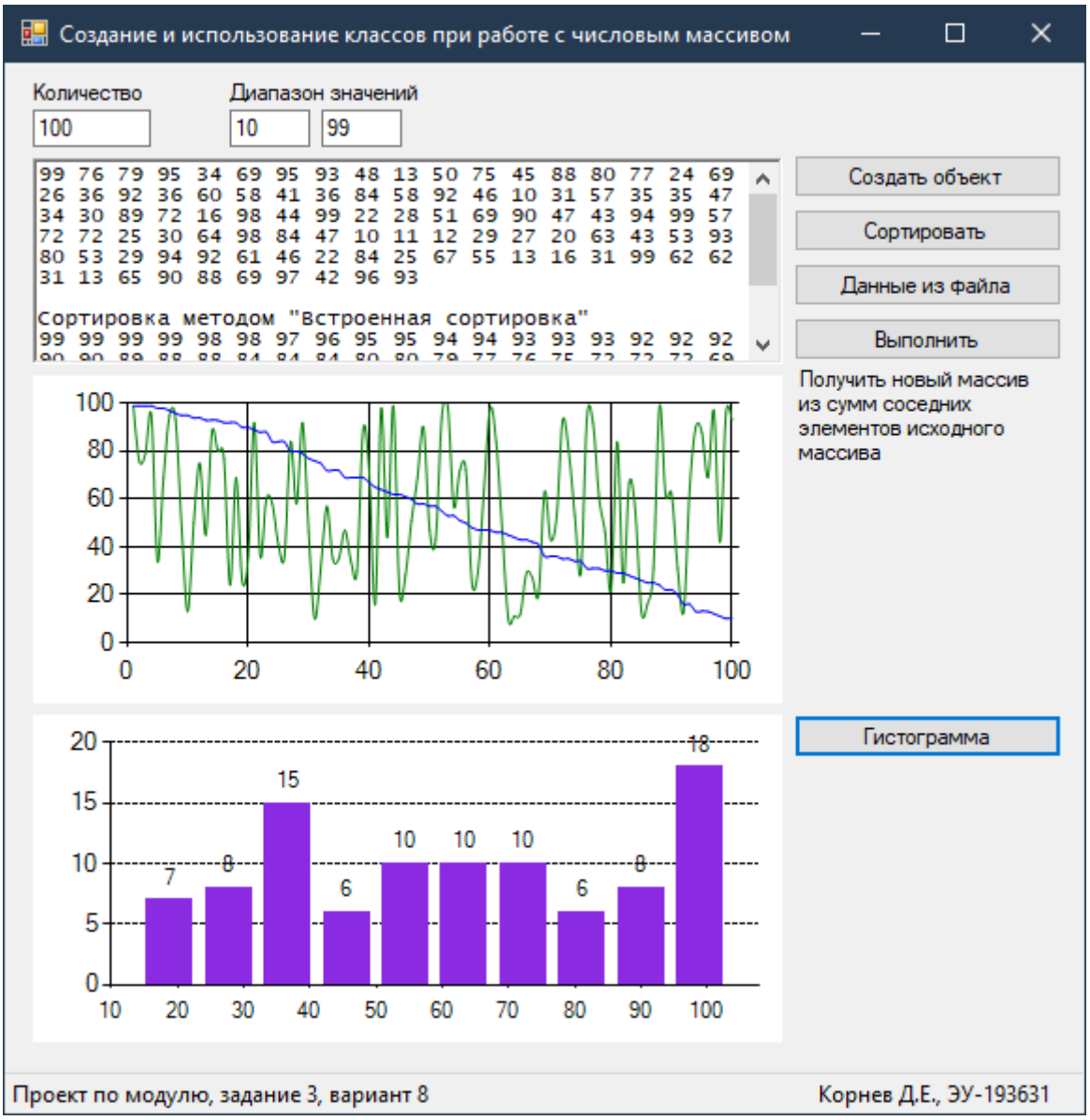

Рисунок 14 - Внешний вид формы Form1.cs

При создании пользовательского интерфейса приложения использовались следующие элементы Windows Forms:

- MaskedTextBox – для ввода количества элементов верном формате (в форме – `elementsCount`);
- TextBox – для ввода диапазона значений (в форме -- `lowerBoundBox` и `upperBoundBox`);

- RichTextBox – для вывода массива и результатов операций с ними;
- Button – для вызова различных функции в форме, например, создания нового массива и его сортировки (в форме – `createButton`, `sortButton`, `readFromFileButton`, `plotHistogramButton` и `executeButton`);
- Label – для отображения пользователю различных пояснений;
- Chart – для визуализации массива в виде графиков Spline и численных интервалов в виде гистограммы;
- StatusStrip – для вывода информации о номере, варианте и авторе работы с помощью двух текстовых элементов.

Далее будет рассмотрено содержание и назначение каждого из модулей программы.

Модуль ArrayKit.cs

В этом модуле находятся три класса. Родительский `CustomArrayBase` описывает целочисленный массив в качестве защищённого поля и различные базовые операции с ним в виде методов. Свойствами класса определены `Size` (размер массива) и `SortMode` (способ сортировки).

```
protected int[] array;
protected int Size { get; set; }
public string SortMode { get; protected set; }
```

Свойства `lowerBound` и `upperBound` будут необходимы при задании произвольных границ массива, или их определении при чтении массива из файла:

```
public int lowerBound { get; protected set; }
public int upperBound { get; protected set; }
```

В этом классе так же описан стандартный конструктор, который хоть и не используется в приложении, но может быть полезен для тестирования базовых функций класса – он инициализирует массив десятью заранее заданными числами и записывает в свойство Size его размер.

```
public CustomArrayBase()
{
    array = new int[] {5, 8, 10, 12, 24, 11, 9, 5, 8, 1};
    Size = array.Length;
}
```

Метод `GetArray()` возвращает массив, объявленный в качестве поля класса `array`:

```
public int[] GetArray()
{
    return array;
}
```

Метод `Sort()` с ключевым словом `virtual` (делает метод доступным для последующего переопределения) записывает в свойство `SortMode` способ сортировки массива (в данном случае – встроенная сортировка из средств стандартного класса `Array`) и сортирует массив этим способом:

```
public virtual void Sort()
{
    SortMode = "Встроенная сортировка";
    Array.Sort(array);
    Array.Reverse(array);
}
```

Метод NeighbouringSum() отвечает за выполнение задания по варианту: находит суммы соседних элементов массива, на основе которых создаёт ещё один массив и возвращает его:

```
public int[] NeighbouringSum()
{
    int[] neighbouringSum = new int[Size - 1];

    for (var i = 0; i < Size - 1; i++)
    {
        neighbouringSum[i] = array[i] + array[i + 1];
    }

    return neighbouringSum;
}
```

В модуле ArrayKit.cs также присутствуют два дочерних класса, которые уже непосредственно используются в приложении.

Первый такой класс – `CustomArray`, унаследованный от `CustomArrayBase`. В этом классе конструктор уже имеет три параметра – размер массива и две его границы. На их основе генерируется новый массив:

```
public CustomArray(int size, int low, int high)
{
    Size = size;
    lowerBound = low;
```

```
        upperBound = high;
        array = RandomArray();
    }
```

Новый метод `RandomArray()` отвечает непосредственно генерацию массива на основе переданных в конструктор значений:

```
    private int[] RandomArray()
    {
        var random = new Random();
        int[] randomArray = new int[Size];

        for (var i = 0; i < Size; i++)
        {
            randomArray[i] = random.Next(lowerBound, upperBound + 1);
        }

        return randomArray;
    }
```

Второй класс – `CustomArrayFromFile`, который позволяет прочитать входной поток из заданного файла и создать массив на его основе. Здесь добавляется новое свойство `FilePath` с открытым `get` и закрытым `set`:

```
    public string FilePath { get; private set; }
```

В конструкторе всего один параметр – путь к файлу:

```
    public CustomArrayFromFile(string path)
    {
        FilePath = path;

        var stream = new StreamReader(FilePath).ReadLine().Split();
        Size = stream.Length;

        array = new int[Size];

        for (var i = 0; i < Size; i++)
        {
            array[i] = Convert.ToInt32(stream[i]);
        }
    }
```

Далее в классе переопределяется метод Sort(), чтобы массив, прочитанный из файла, сортировался уже не встроенным способом, а методом Selection2. Меняется значение свойства SortMode, и производится сортировка:

```
    override public void Sort()
```

```
{
    SortMode = "Selection2";

    // сортировка методом Selection2
    int i = 0, max, nmax;
    while (i < Size - 1)
    {
        max = array[i];
        nmax = i;
        // цикл внутренних итераций
        int j = i + 1;
        while (j < Size)
        {
            if (array[j] > max)
            {
                max = array[j];
                nmax = j;
            }
            j++;
        }
        // сохранение текущего элемента
        array[nmax] = array[i];
        array[i] = max;
        i++;
    }
}
```

Модуль Form1.cs

Данный модуль состоит из одного класса `Form1`, отвечающего за все необходимые действия с формой. В классе объявлены четыре поля: экземпляр базового класса `CustomArrayBase`, величина массива и его границы:

```
CustomArrayBase array;
int size = 100, low = 10, high = 99;
```

Также для удобства обращения к объектам `Chart` были введены вспомогательные переменные `MainSpline` и `SortedSpline`, обозначающие серии графиков:

```
string MainSpline = "Series1", SortedSpline = "Series2";
```

Конструтор формы не выполняет никаких действий, кроме её инициализации:

```
public Form1()
{
    InitializeComponent();
}
```

Метод логического типа `InputErrorBuster()` проверяет введённые значения на предмет ошибок, возвращая истину в случае их отсутствия и ложь в случае обнаружение какого-либо несоответствия типов:

```
private bool InputErrorBuster()
{ // проверка ошибок ввода
    try
    {
        size = Convert.ToInt32(elementsCount.Text);
        low = Convert.ToInt32(lowerBoundBox.Text);
        high = Convert.ToInt32(upperBoundBox.Text);
    }
    catch
    {
        MessageBox.Show("Неверные данные на входе.\nПроверьте введённые
                         значения", "Ошибка ввода");
        return false;
    }

    if (low > high)
    {
        MessageBox.Show("Нижняя граница не может быть больше верхней",
                        "Ошибка ввода");
        return false;
    }

    return true;
}
```

Метод RefreshOutput() был введён для исключения повторений в коде однотипных действий – очистки области вывода и графиков и вывода новой информации. Он вызывается при каждой новой генерации массива или чтении из файла:

```
private void RefreshOutput()
{
    outputBox.Clear();
    outputBox.AppendText(String.Join(" ", array.GetArray()));
    chart1.Series[SortedSpline].Points.Clear();
    chart1.Series[MainSpline].Points.DataBindY(array.GetArray());

    sortButton.Enabled = true;
    executeButton.Enabled = true;
}
```

Обработчик кнопки «Создать» инициализирует экземпляр класса `CustomArrayBase` с помощью конструктора его потомка `CustomArray`, таким образом, обозначая, что далее необходимо работать с массивом с задаваемым размером и границами. Перед

инициализацией происходит проверка ошибок, после инициализации – массив выводится с помощью метода `RefreshOutput()`:

```
private void createObjectButton_Click(object sender, EventArgs e)
{ // обработчик кнопки "Создать"
    if (!InputErrorBuster()) return;

    array = new CustomArray(size, low, high);

    RefreshOutput();
}
```

Обработчик кнопки «Сортировать» вызывает метод `Sort()` из экземпляра `array`, а затем выводит уже отсортированный массив в область вывода, после чего строится график в отдельной серии:

```
private void sortButton_Click(object sender, EventArgs e)
{ // обработчик кнопки "Сортировать"
    array.Sort();
    outputBox.AppendText($"\n\nСортировка методом \"{array.SortMode}\"\n"
                         + String.Join(" ", array.GetArray()));

    chart1.Series[SortedSpline].Points.DataBindY(array.GetArray());
}
```

Обработчик кнопки «Данные из файла» инициализирует (или реинициализирует) экземпляр array дочерним классом CustomArrayFromFile и выводит массив, получив путь к файлу, задаваемый в качестве параметра в конструкторе, из диалога открытия файла. Если в ходе выбора/чтения файла возникли ошибки или файл не был выбран – программа не будет создавать массив, а просто выйдет из обработчика:

```
private void readFromFileButton_Click(object sender, EventArgs e)
{ // обработчик кнопки "Данные из файла"
    string fileName;

    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    { // сохранение имени файла
        fileName = openFileDialog1.FileName;
    }
    else return;

    try
    { // проверка верности данных в массиве
        array = new CustomArrayFromFile(fileName);
    }
```

```
        catch
        {
            MessageBox.Show($"Данные в файле {fileName} не могут быть
                              использованы для создания массива.\n" +
                            $"Убедитесь, что числа записаны в одной строке
                              через пробел",
                            "Ошибка чтения файла");

            return;
        }

        RefreshOutput();
    }
```

Обработчик кнопки «Выполнить» выводит значения массива, подаваемые из метода `NeighbouringSum()` экземпляра `array` – суммы соседних элементов оригинального массива:

```
    private void executeButton_Click(object sender, EventArgs e)
    { // обработчик кнопки "Выполнить"
        outputBox.AppendText($"\n\nМассив сумм соседних элементов
                               оригинального массива\n"
                             + String.Join(" ", array.NeighbouringSum()));
    }
```

Обработчик кнопки «Гистограмма» строит гистограмму в объекте `chart2`, используя логику, описанную в методе `Plot()` статического класса `Histogram` (о нём речь пойдёт позже):

```
    private void plotHistogramButton_Click(object sender, EventArgs e)
    { // обработчик кнопки "Гистограмма"
        var count = (int)histogramIntervalsUpDown.Value;
        double[] x;
        int[] y;

        Histogram.Plot(array.GetArray(), count, out x, out y);

        chart2.Series["Series1"].Points.DataBindXY(x, y);
    }
```

Метод TextBoxKeyPressRule() отвечает за обработку нажатий клавиш в необходимых полях. Он установлен как обработчик событий в полях ввода диапазона, чтобы избежать ввода посторонних символов. В качестве «разрешенных» установлены знак «-» и все цифры:

```
private void TextBoxKeyPressRule(object sender, KeyPressEventArgs e)
{
    if (e.KeyChar != '\b' && e.KeyChar != '-')
        e.Handled = !Char.IsDigit(e.KeyChar);
}
```

Модуль Histogram.cs

Этот модуль содержит статический класс Histogram с одним методом Plot(), в котором описана логика построения гистограммы в форме. Особенность её построения в том, что необходимо разбить текущий массив чисел на равные интервалы и посчитать количество попаданий в них. На вход подается массив значений и количество интервалов, также есть два выходных параметра:

```
public static void Plot(int[] values, int count, out double[] x, out int[] y)
```

Выходные параметры – это массивы границ интервалов и количества попаданий:

```
x = new double[count];  // массив границ интервалов
y = new int[count];     // массив попаданий в интервалы
```

Сначала находятся минимальный и максимальный элементы массива, а затем вычисляется длина интервала:

```
// интервалы
int min = (int)1e6;
int max = (int)-1e6;
for (var i = 0; i < values.Length; i++)
{ // нахождение минимума/максимума для границ интерваов
    if (values[i] > max)
    {
        max = values[i];
    }
    if (values[i] < min)
    {
        min = values[i];
    }
}

var intervalLength = (double)(max - min) / count;
```

Затем в массив x записываются границы интервалов:

```
for (var i = 1; i < count; i++)
{ // расстановка интервалов
    x[i - 1] = min + intervalLength * i;
}
```

```
x[count - 1] = max;
```

А в массив y – попадания в интервалы:

```
for (var i = 0; i < values.Length; i++)
{
    if (values[i] > min && values[i] <= x[0])
    { // диапазон [min; min + intervalLength]
        y[0]++;
    }
    else
    { // последующие диапазоны
        for (var j = 1; j < count; j++)
        {
            if (values[i] > x[j - 1] && values[i] <= x[j])
            {
                y[j]++;
            }
        }
    }
}
```

После завершения работы метода массивы x и y подаются на выход и обновляются вне метода, что позволяет построить гистограмму.

3.3. Диаграмма классов

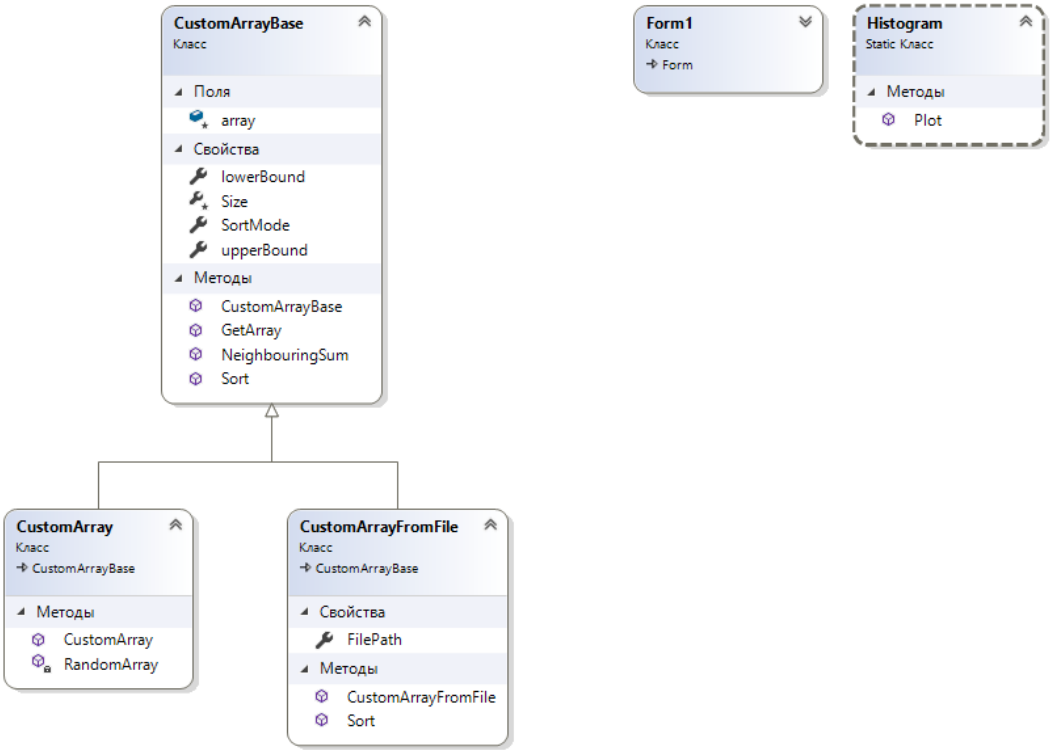

Рисунок 15 - Развёрнутая диаграмма классов проекта one-dim-array

3.4. Тестирование

Так как большинство ошибок ввода «отсечено» обработкой нажатия клавиш в текстовых полях и маской, проверить работоспособность обработчика исключений можно, оставив поля пустыми.

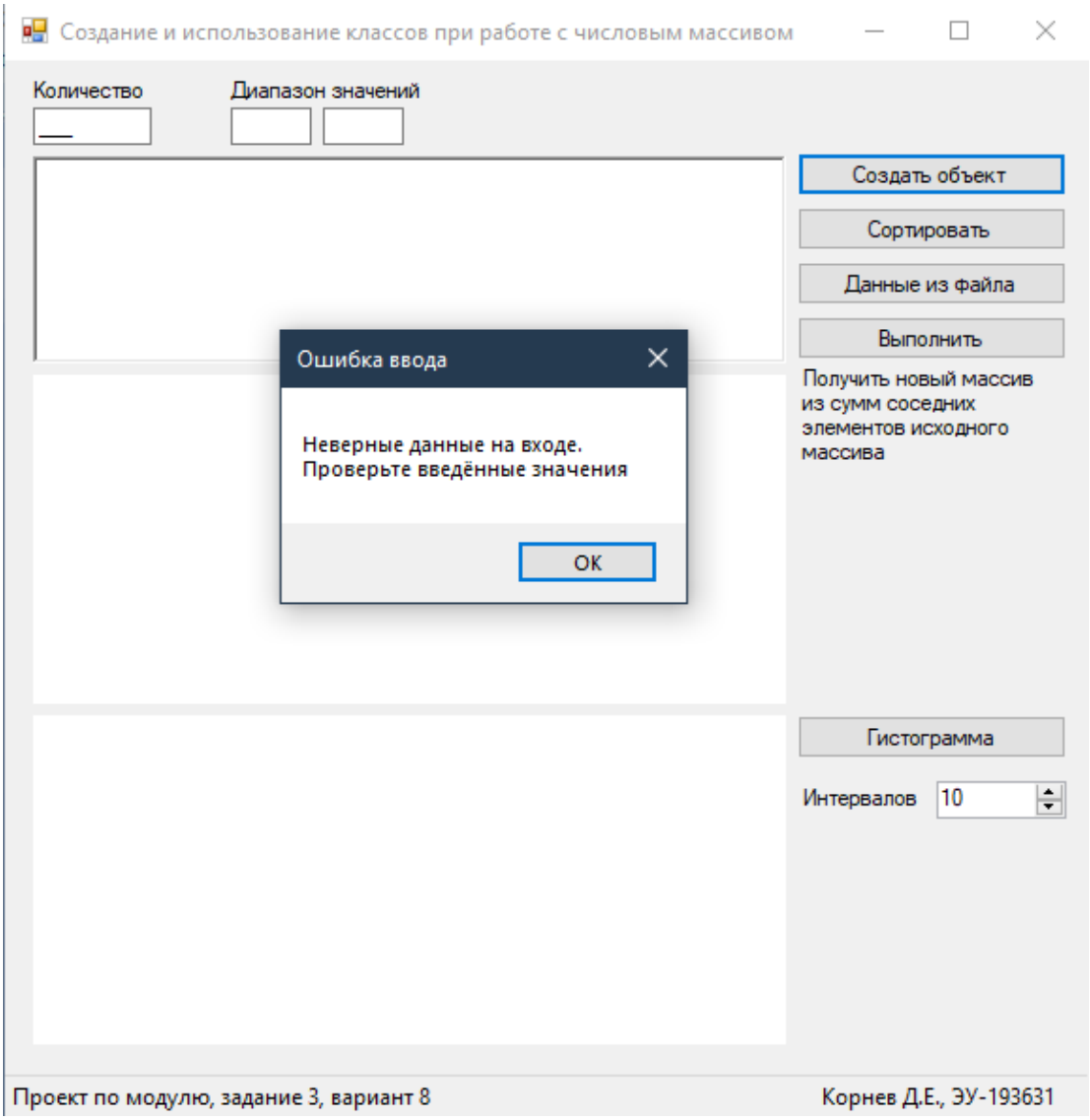

Рисунок 16 - Обработанное исключение при ошибке ввода

Если ввести нижнюю границу, превосходящую верхнюю, получим следующую ошибку:

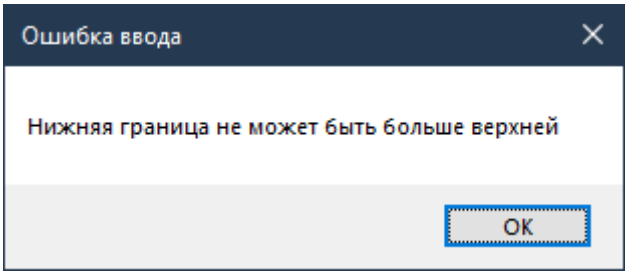

Рисунок 17 - Ошибка ввода границ массива

Для проверки функции чтения из файла я создал два текстовых файла с названиями good_array.txt и bad_array.txt. В первом я сохранил массив, записанный в верном формате – в одной строке через пробел, во втором также сохранил массив, но «испортил его»: среди чисел также добавились буквы, лишние пробелы и т.д.

При чтении «испорченного» файла программа обнаружила, что массив хранится в неверном формате, поэтому обработала данную ошибку:

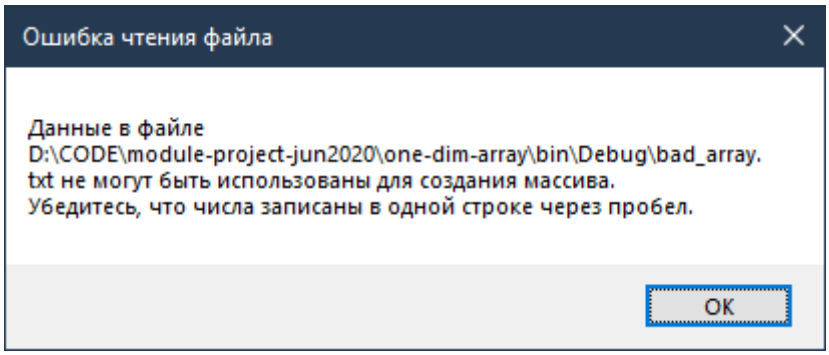

Рисунок 18 - Ошибка чтения файла с неверным массивом

Но необходимо также протестировать и правильность работы программы – ввести все значения правильно. Если это сделать, программа работает прекрасно:

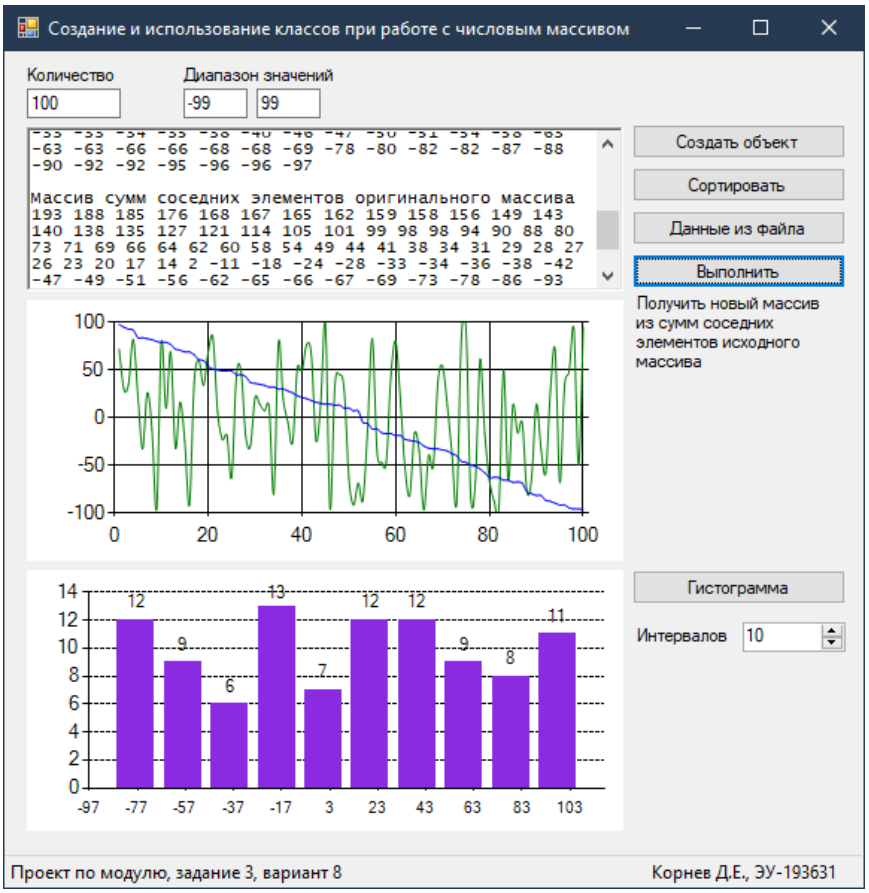

Рисунок 19 - Приложение в рабочем состоянии

Из файла good_array.txt данные также читаются без ошибок:

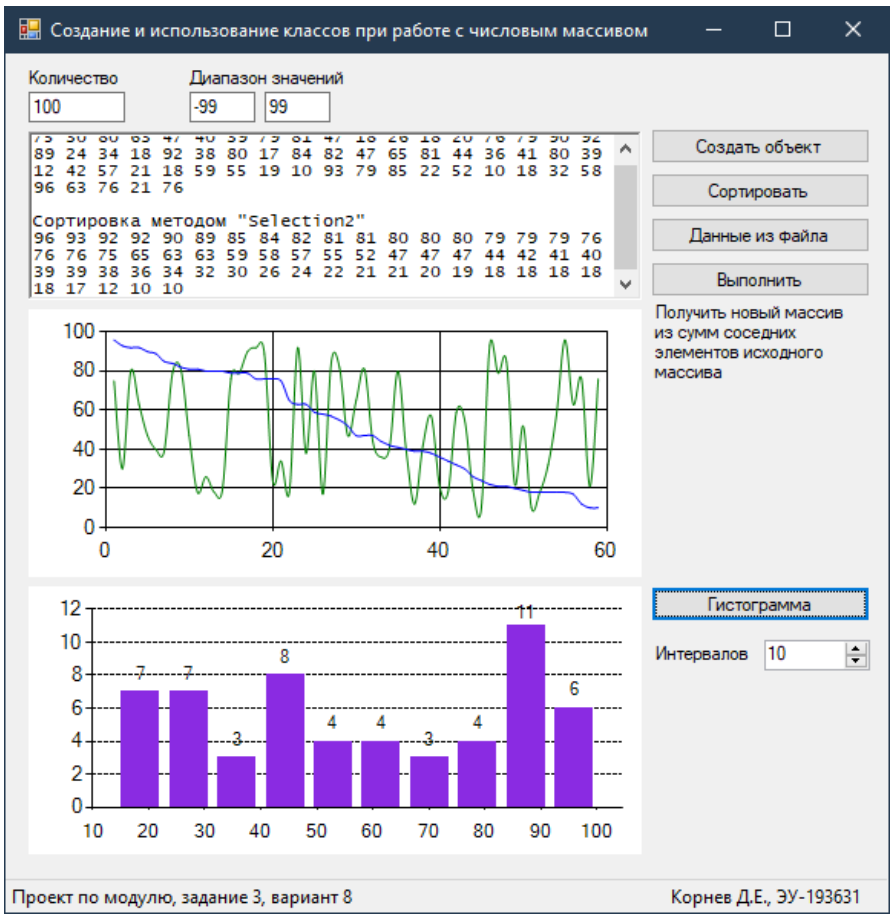

Рисунок 20 - Прочитанный и отсортированный массив из файла

3.5. Вывод

При создании данного приложения я воспользовался принципами и преимуществами объектно-ориентированного программирования, закрепил навыки создания, наследования и использования классов, управления уровнями доступа к данным. Также были применены знания об обработке событий формы, построении графиков с помощью элемента Chart, алгоритмах сортировки и форматировании вывода.

Заключение

В данном проекте были освещены основные черты структурного и объектно-ориентированного подхода к разработке программного обеспечения и приведена их сравнительная характеристика. Также были разработаны два приложения на языке C#: для приближённого вычисления интеграла заданных функций и для работы с одномерным числовым массивом.

В ходе выполнения заданий проекта по модулю были закреплены и применены на практике навыки, полученные при обучении на дисциплинах «Программирование» и «Объектно-ориентированный анализ и программирование» в 1 и 2 семестрах 1 курса. Главным результатом выполнения этой работы можно считать усвоение материала этих дисциплин и его понимание на необходимом уровне.

Список использованных источников

1. **Объектно-ориентированный анализ и проектирование с примерами приложений** [Книга] / авт. Гради Буч Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен, Келли А. Хьюстон. - [б.м.] : "Вильямс", 2010.

2. **Go To Statement Considered Harmful** [Журнал] / авт. Dijikstra E.. - [б.м.] : Association for Computing Machinery, Inc., 3 Март 1968 г.. - Т. 11. - стр. 147-148.

3. **Typeful Programming** [Книга] / авт. Карделли Лука. - New York : IFIP State-of-the-Art Reports, 1991.

4. **О структурном программировании / Хабр** [В Интернете] // Хабр. - 22 Февраль 2011 г.. - Июнь 2020 г.. - https://habr.com/ru/post/114326/.

5. **Обзор Объектно-Ориентированного Программирования** [В Интернете] // First Steps / ред. Хромов В.. - Июнь 2020 г.. - http://www.firststeps.ru/theory/oop/.

6. **Современный взгляд на концепцию структурного программирования** [Журнал] / авт. Авачева Т. Г. Пруцков А. В.. - 2019 г.. - 4 : Т. 6. - стр. 646-665.

7. **Структурное проектирование и конструирование программ** [Книга] / авт. Йодан Э.. - [б.м.] : Мир, 1979. - стр. 174-175.