МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Тульский государственный университет»

Институт прикладной математики и компьютерных наук

Кафедра вычислительной техники

Курсовая работа по дисциплине

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

на тему: Исследование вычислительной эффективности объектно-ориентированных приложений.

Студент группы \_\_\_220681\_\_\_ Шайхаттаров Д.В. \_\_\_\_\_\_\_\_\_\_\_

(Ф.И.О.) (Подпись, дата)

Руководитель работы к.т.н., доц. Берсенев Г.Б. \_\_\_\_\_\_\_\_\_\_\_

(Ф.И.О., должность) (Подпись, дата)

Комиссия: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Тула, 2020

МИНОБРНАУКИ РОССИИ

ФГБОУ ВО «Тульский государственный университет»

Институт прикладной математики и компьютерных наук

Кафедра «Вычислительная техника»

**З А Д А Н И Е**

На курсовую работу по дисциплине «Объектно-ориентированное программирование»

студенту группы 220681 Шайхаттарову Дамиру Владимировичу

Тема работы:

«Исследование вычислительной эффективности объектно- ориентированных приложений»

Входные данные Вариант №19:

задача: определение площади методом Монте-Карло;

фигура: dego;

приложение №1: процедурное приложение на базе Console/C#;

приложение №2: объектно-ориентированное приложение на базе Console/C#.

Задание получил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ «22» февраля 2020.

(подпись студента)

Срок предоставления задания «01» июня 2020.

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (расшифровка подписи)

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_г.

К защите. Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись) (расшифровка подписи)

Замечания руководителя \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_г.

*При защите курсового проекта (работы) наличие рецензии обязательно.*

**СОДЕРЖАНИЕ**[ВВЕДЕНИЕ 4](#_Toc54012839)

[1. ПОСТАНОВКА ЗАДАЧИ 6](#_Toc54012840)

[2. РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ 9](#_Toc54012841)

[2.1. Анализ вариантов использования программы 9](#_Toc54012842)

[2.2. Требования к приложению 10](#_Toc54012843)

[3. АНАЛИЗ АЛГОРИТМА РЕШЕНИЯ 12](#_Toc54012844)

[3.1. Метод Монте-Карло 12](#_Toc54012845)

[3.2. Определение попадания точки внутрь фигуры 12](#_Toc54012846)

[4. ПРОЦЕДУРНОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C# 14](#_Toc54012847)

[4.1. Программные средства разработки 14](#_Toc54012848)

[4.2. Логическое проектирование 14](#_Toc54012849)

[4.3. Системные требования 14](#_Toc54012850)

[4.4. Структура приложения 15](#_Toc54012851)

[4.5. Структура приложения 18](#_Toc54012852)

[4.6. Результаты работы 19](#_Toc54012853)

[5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C# 20](#_Toc54012854)

[5.1. Программные средства разработки 20](#_Toc54012855)

[5.2. Логическое проектирование 20](#_Toc54012856)

[5.3. Описание программы 22](#_Toc54012857)

[5.4. Руководство системного программиста 31](#_Toc54012858)

[5.5. Руководство программиста 33](#_Toc54012859)

[5.6. Руководство оператора 34](#_Toc54012860)

[5.7. Результаты работы 36](#_Toc54012861)

[6. АНАЛИЗ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ ПРИЛОЖЕНИЙ 38](#_Toc54012862)

[7. УЛУЧШЕНИЕ ХАРАКТЕРИСТИК КАЧЕСТВА ПРИЛОЖЕНИЙ 39](#_Toc54012863)

[7.1. Использование встроенной оптимизации кода 39](#_Toc54012864)

[7.2. Профилирование 39](#_Toc54012865)

[7.3. Результаты улучшения характеристик 41](#_Toc54012866)

[ЗАКЛЮЧЕНИЕ 42](#_Toc54012867)

[СПИСОК ЛИТЕРАТУРЫ 43](#_Toc54012868)

[ПРИЛОЖЕНИЕ 44](#_Toc54012869)

[Файл Program.cs (процедурное приложение) 44](#_Toc54012870)

[Файл Program.cs (объектно-ориентированное приложение) 48](#_Toc54012871)

[Файл Menu.cs 48](#_Toc54012872)

[Файл MonteCarloCalculator.cs 52](#_Toc54012873)

[Файл Triangle.cs 53](#_Toc54012874)

[Файл Quadrant.cs 54](#_Toc54012875)

[Файл Side.cs 55](#_Toc54012876)

[Файл Point.cs 55](#_Toc54012877)

# ВВЕДЕНИЕ

В настоящее время объектно-ориентированное программирование (ООП) является доминирующим стилем при создании больших программ и программных систем. Процедурно-ориентированное программирование, широко использовавшееся до появления ООП, обычно позволяет создавать более эффективные в вычислительном отношении реализации приложений, что является существенным фактором при разработке систем реального времени. На практике эти два стиля программирования часто используются совместно, позволяя варьировать степень их применения в программах.

Использование объектно-ориентированного (ОО) подхода при разработке программного обеспечения позволяет преодолеть естественную сложность разрабатываемого ПО, упростить процесс отладки и последующего сопровождения, расширения и переноса ПО на другие платформы.

ОО подход включает в себя объектно-ориентированный анализ (ООА), дизайн (проектирование) (ООД) и программирование.

Объектно-ориентированный анализ – это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Объектно-ориентированное проектирование – это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, статической и динамической моделей проектируемой системы.

Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. Идеальное ОО приложение должно быть расширяемым, масштабируемым, сопровождаемым и переносимым.

Основным понятием ООП является класс. Класс (class) определяет группу объектов с общими свойствами (атрибутами), поведением (функциями), семантикой и связями с другими объектами. Класс можно трактовать как шаблон для создания объектов. Каждый объект является экземпляром некоторого класса, причем только одного. Класс может наследовать один или нескольких интерфейсов, реализуя свойства, события и методы каждого из них.

Основными концепциями ООП, которыми руководствуются при создании классов, являются инкапсуляция, наследование и полиморфизм (параметрический и основной, применяемый при наследовании). При создании сложных объектов наряду с наследованием (отношением «is-a») широко используется включение объектов (отношение «is-part-of»)).

Существует огромное количество методологий и рекомендаций, направленных на повышение эффективности процесса проектирования программных систем. Среди них можно выделить принципы SOLID для гибкого проектирования объектно-ориентированного ПО.

C# – объектно-ориентированный язык программирования. Он относится к семье языков с C-подобным синтаксисом. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

В данной курсовой работе создаются процедурно-ориентированная и объектно-ориентированная реализации конкретного приложения и проводится сравнительный анализ их вычислительной эффективности.

# ПОСТАНОВКА ЗАДАЧИ

В результате выполнения курсовой работы необходимо создать и сравнить по вычислительной эффективности два приложения, решающие задачу приближенного вычисления площади геометрической фигуры методом Монте-Карло с использованием процедурного и объектно-ориентированного программирования соответственно. Также нужно выполнить исследование вычислительной эффективности созданных приложений и представить результаты в виде таблиц, провести анализ и улучшить заданные характеристики качества приложений.

Вариант задания №19:

* геометрическая фигура: *dego*;
* приложение1: процедурное приложение на базе Console/C#;
* приложение2: объектно-ориентированное приложение на базе Console/C#;

Геометрическая фигура представляет собой четверть круга с центральной точкой *o* и радиусом *od* и треугольник с вершинами *e, g, o* (рис. 1).

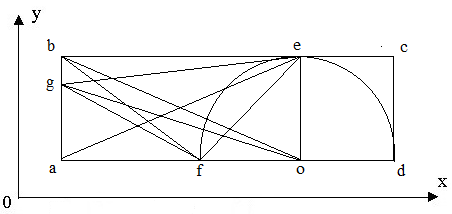


Рис. 1. Фигура *dego*

Исходными данными для каждого приложения являются координаты точек *a, e, g*. Для определения однозначного определения расположения фигуры необходимо задать 2 точки *d, o*. После чего определить координаты точки *с*, чтобы, используя в том числе и точку *a*, определить расположение и размер прямоугольника *abcd*. Далее генерируем случайным образом точки, которые должны равномерно заполнять площадь прямоугольника. Для этого случайные числа должны иметь равномерное распределение (по ширине и высоте прямоугольника соответственно).

Для каждой точки выполняется проверка, попала ли точка внутрь заданной фигуры. Если из *N* точек *M* точек оказалось внутри фигуры, а площадь прямоугольника равна *S*, то площадь фигуры будет приближенно равна .

Поскольку площадь фигуры легко определяется по правилам геометрии, мы можем определить относительную погрешность приближенного вычисления этой площади методом Монте-Карло. Естественно, чем больше *N*, тем меньше погрешность такого вычисления.

Каждое приложение шесть раз повторяет эксперимент и вычисляет площадь фигуры методом Монте-Карло для *N =* 103, 104, 105, 106, 107 соответственно. В каждом эксперименте определяется относительная погрешность вычисления площади (в процентах) и его длительность (в миллисекундах).

По результатам экспериментов каждое приложение выдает на экран таблицу, показывающую зависимости значений относительной погрешности и длительности эксперимента от величины *N*.

Задачами курсовой работы являются:

* приобретение навыков решения вычислительных задач;
* практическое освоение современных инструментальных систем разработки ПО;
* сравнительный анализ вычислительной эффективности процедурных и объектно-ориентированных программ;
* получение навыков создания программ на языке C# платформы Microsoft .NET Core;
* приобретение практических навыков оформления и выпуска документации в соответствии с требованиями стандартов (ЕСПД, UML).

# РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ

## 2.1. Анализ вариантов использования программы

Необходимо разработать программу, которая вычисляет площадь фигуры методом Монте-Карло. Программа должна предоставить пользователю возможность выполнения следующих действий:

* ввод координат (вручную или из контрольного примера);
* нахождение площади фигуры методом Монте-Карло;
* вывод результатов на экран;
* ввод новых значений;
* повторный расчёт;
* выход.

Диаграмма вариантов использования приложений представлена на рис. 2. В диаграмме были использованы такие виды отношений как расширения (пунктирная стрелка с подписью расширить), включение (пунктирная стрелка с подписью включить) и наследование (стрелка с не закрашенным треугольником). Расширение означает, что процедура может выполняется в зависимости от некоторых условий. Включение указывает на то, что одна процедура включается в некоторой точке в другую в качестве составного компонента. Наследование представляет собой связь между родителем и потомком, который наследует все его свойства.



Рис. 2. Диаграмма вариантов использования приложений

## 2.2. Требования к приложению

2.2.1. Общие требования

Каждое приложение должно соответствовать следующим требованиям:

* тип: настольное приложение;
* взаимодействие с приложениями производится через консоль;
* язык программирования: C#;
* поддержка следующих пунктов меню: ввод, ввод контрольного примера с последующим расчётом, вывод текущих входных данных, расчёт, выход;
* возможность получения координат из консоли, куда пользователь будет вводить значения;
* координаты представляют собой пару значений *x* и *y* для каждой точки фигуры;
* проверка корректности вводимых данных на возможность построить на основе вводимых данных построить фигуру *dego*;
* после приложение должно выводить в консоль количество точек N, точную площадь фигуры, площадь фигуры, вычисленную по методу Монте-Карло, относительную погрешность и время за которое были совершены вычисления для *N =* 103, 104, 105, 106, 107.

2.2.2. Требования к процедурному приложению

Каждое процедурное приложение должно соответствовать следующим требованиям:

* Фигура должна описываться с помощью точек, заданных координатами *x* и *y*;
* Все методы приложения должны располагаться в одном статическом классе;
* Используется процедурная парадигма программирования.

2.2.3. Требования к объектно-ориентированному приложению

Каждое объектно-ориентированное приложение должно соответствовать следующим требованиям:

* Фигура должна описываться с помощью предназначенных для этого классов (Triangle и Quadrant);
* Все методы, участвующие в поиске площади отдельной фигуры должны располагаться в классе этой фигуры. Работа с этими методами должна производится через экземпляр этого класса;
* Каждый класс должен иметь три типа конструкторов: по умолчанию, инициализации, копии;
* Используется объектно-ориентированная парадигма программирования.

# АНАЛИЗ АЛГОРИТМА РЕШЕНИЯ

Пусть координаты точек заданной фигуры (рис. 3) объявленны следующим образом: *d(dx, dy), e(ex, ey), g(gx,xy), o(ox, oy)*.

Координаты точек *a, e, g* задаются пользователем, а координаты остальных точек определяются следующим образом :

;

*;*

*.*

## 3.1. Метод Монте-Карло

Впишем исходную фигуру в прямоугольник , площадь которого:

.

Сгенерируем в этом прямоугольнике *N* точек, координаты которых выбираются случайным образом.

Определим число точек *M*, которые попали в фигуру.

Приближённую площадь фигуры вычислим по следующей формуле:

Для нахождения точной площади треугольника используется формула Герона:

,

где *p* – полупериметр; *a*, *b*, *c* – длины сторон треугольника:

;

, по этой же формуле можно посчитать длинны сторон *b* и *с*

Для нахождения погрешности вычислений в процентах методом Монте-Карло можно воспользоваться следующей формулой:

.

Для нахождения точной площади квадранта используем формулу:

;

Где R – расстояние между точками *o* и *d* и т.к. *od* параллельно оси *X*, то

.

## 3.2. Определение попадания точки внутрь фигуры

Необходимо сгенерировать внутри прямоугольника *abec* точку и определить, принадлежит ли эта точка фигуре *dego*, если известны координаты точек фигуры. Пример расположения точек фигуры и точки *p* представлен на рис. 3.

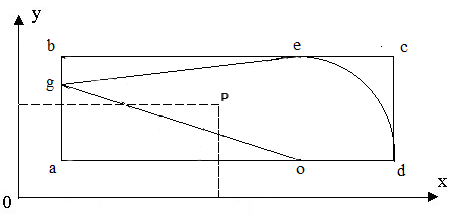


Рис. 3. Точки, необходимые для расчётов

Для определения попадания точки внутрь треугольника используется следующий векторное и псевдоскалярное произведения:

Где индексы 1, 2, 3 указывают на то, что координаты относятся к вершинам треугольников, а 0 на точку, проверяемую на попаданию в треугольник. Если они одинакового знака, то точка внутри треугольника, если результат хотя-бы одной формулы - ноль, то точка лежит на стороне, иначе точка вне треугольника.

А для определения попадания точки внутрь квадранта используется формула окружности: , где *x, y* – это координаты точки, проверяемой на попадание в квадрант, а *a, b* – координаты точки центра окружности, а *R* – радиус окружности (*od*).

В данном случае мы имеем дело с квадрантом, из чего следует, что, помимо условия,нам потребуется ограничить нашу область двумя прямыми od и oe, то есть требуемыемыми условиями будут: и .

# ПРОЦЕДУРНОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C#

## 4.1. Программные средства разработки

Процедурное приложение выполняется в инструментальной системе MS Visual Studio 2019. Она предоставляет системную объектно-ориентированную среду на базе платформы Microsoft .NET Core для разработки настольных (клиентских) приложений. Для разработки приложения используется предоставленный платформой класс System::Console для создания консольных приложений.

## 4.2. Логическое проектирование

Все методы, которые используются для нахождения площади, расположены в основном классе Program, в котором также располагается основной метод Main. и остальные методы необходимые для решения нашей задачи, точки задаются пользователем или с помощью примера, также необходимо упомянуть, что точки задаются через переменные, обозначающие x и y координаты необходимых точек

## 4.3. Системные требования

Для корректной работы приложения требуется:

* Операционная система Windows 7 и выше;
* платформа .NET Core 3.1 и выше;
* архитектура 32-разрядная (х86)/64-разрядная (х64);
* оперативная память 512 МБ и более;
* устройства ввода: клавиатура и мышь.

## 4.4. Структура приложения

Диаграмма класса Program представлена на рис. 4.



Рис. 4. Диаграмма класса Program

Приложение одержит следующие функции:

static void Main() – здесь запускается цикл за счёт которого работает меню, используя метод static void DrawMenu(), который отвечает за то, как оформление меню, и метод static string ReadInputCommand() – , который необходим для управления меню. Используя пункты меню: «Calculate», работающий с исходными данными введёнными пользователем и «Show an Example», использующий данные из заранее заготовленного примера, приложение вызывает метод public static void MonteCarlo(), принимающий координаты точек *a, e ,g* и реализующий расчёт и вывод расчётов в консоль в виде таблицы. Чтобы самостоятельно ввести данные для метода MonteCarlo() необходимо выбрать пункт меню «Input Data», вызывающий инициализацию координат точек *a, e ,g* за счёт метода static void SetPoints(), принимающий переменные *x, y* координат точек *a, e ,g* с ключевым словом «ref» перед переменной, позволяющий изменять поля метода внутри другого метода, также для инициализации каждой точки используется метод static void SetPoint(), обеспечивающий форматированный ввод координат точки, а обычный Console.Readline(), заменён методом static double CheckedReadLine(), проверящий вводимую строку на то является ли она числом. Для проверки ошибок при инициализации точек используется метод static List<string> ErrorList(), которые возвращает список заполненный ошибками инициализации точек, если таковые присутствуют. Если размер списка не равен нулю, то опять вызывает метод SetPoints(), иначе инициализация будет считаться успешной и в консоль будет выведены текущие координаты фигуры *dego*, используя метод static void ShowCurrentData(), принимающий координаты точек *a, e, g* , также стоит упомянуть, что если данные не вводились и пользователь выберет пункт меню «Calculate», то приложение укажет на отсутствие данных и попросит ввести их и между каждым действием для того, чтобы пользователь мог прочитать все сообщения, выводимые программой, будет вызываться метод static void PressAnyKeyToContinue(), приостанавливающий программу и выводящий сообщение о том, что пользователь может продолжить работу приложения, нажав любую клавишу.

public static void MonteCarlo(double ax, double ay, double ex, double ey, double gx, double gy) – принимает координаты точек a, e, g , достаточные для того, чтобы узнать все координаты заданной фигуры *dego* и прямоугольника, описывающего данную фигуру. В процессе реализации метод использует цикл, каждая итерация которого считает точную площадь, используя метод GetSquare, который использует сумму возвращаемых значений методов TriangleSquare и QuadrantSquare, где также используется метод Length, принимающий координаты точек отрезка длину, которогу мы ищем и соответственно возвращает длину этого отрезка, далее вычисляется и записывается в предназначенный для этого список, площадь, вычесленная методом Монте-Карло, используя метод MonteCarloSquare, который проверяет попали точка в фигуру *dego* для чего использует CheckPointInTriangle и CheckPointInQuadrant, возвращающие истину или ложь в зависимости от того попала ли точка в треугольник или квадрант, и если точка попала в треугольник или квадрант, составляющие фигуру *dego*, счётчик попаданий увеличивается на единицу, далее по формуле Монте-Карло вычисляется площадь фигуры и результат опять же записывается в предназначенный для этого список, также метод MonteCarlo считает относительную погрешность вычислений для каждого набора точек, которые мы получаем во время итерации, путём изменения счётчика. Счётчик имеет начальное значение 103 и конечное 107 и при каждой итерации значение счётчика увеличивается в 10 раз. После чего с помощью цикла, имеющий начальное значение 0, а конечное равное количеству итераций предыдущего цикла, и метода DrawTable строится таблица выходных данных, а также DrawHorizontalLine для вывода вверхней и нижней границы таблицы. Более детально алгоритм можно рассмотреть на рис. 5.

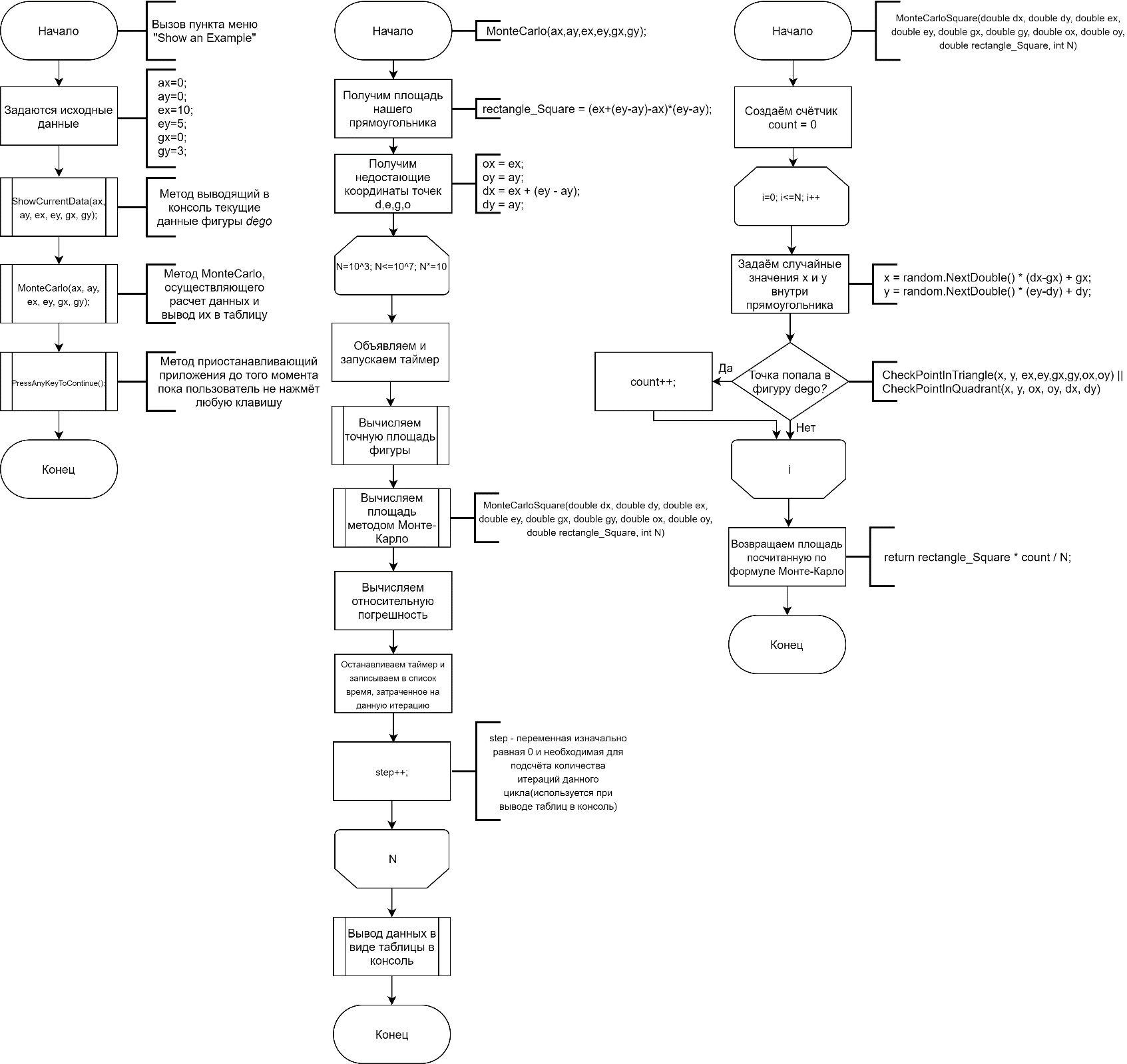


Рис. 5. Схемы метода MonteCarlo и его вызова внутри меню

## 4.5. Результаты работы

Результаты работы приложения представлены на рис. 6.

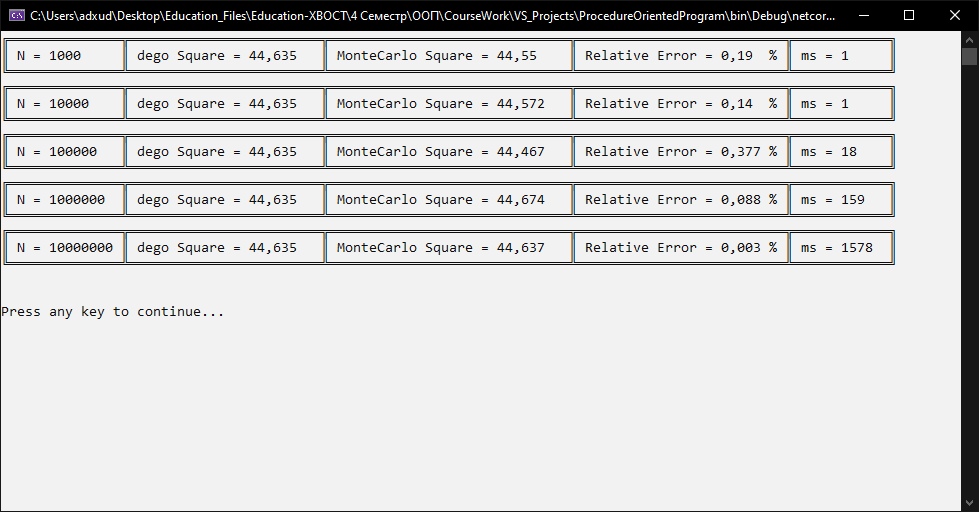


Рис. 6. Тестирование процедурного приложения

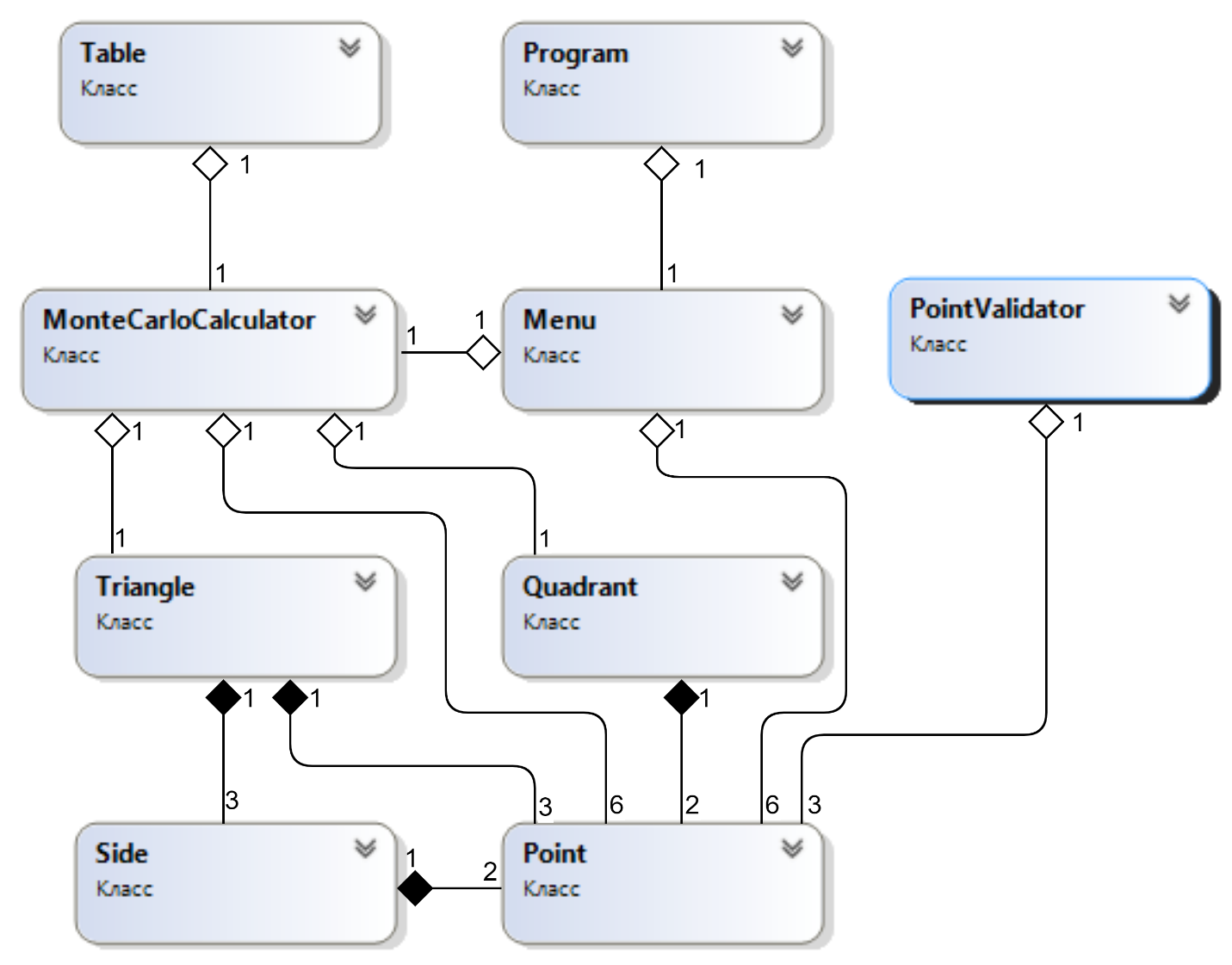
# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C#

## 5.1. Программные средства разработки

Объектно-ориентированное приложение выполняется в инструментальной системе MS Visual Studio 2019. Она предоставляет системную объектно-ориентированную среду на базе платформы Microsoft .NET Core для разработки настольных (клиентских) приложений. Для разработки приложения используется предоставленный платформой класс System::Console для создания консольных приложений.

## 5.2. Логическое проектирование

В рассматриваемой предметной области можно выделить следующие сущности: точка, сторона, треугольник и квадрант. Размеры и форма треугольника задаются тремя точками, а квадранта двумя. Каждая точка задаётся двумя координатами: *х* и *у*. Каждая сторона задаётся двумя точками. Для сущности точки можно создать класс Point, который будет содержать конструкторы и поля *х*, *у*. Для сущности сторона можно создать класс Side, который будет содержать поля p1 и p2(экземпляры класса Point). Для сущности треугольник можно создать класс Triangle, который будет содержать такие поля: три объекта класса Point и три объекта класса Side, таким же образом можно задать Quadrant, только для этого класса будет использоваться 2 экземпляра класса Point, один для центра, другой для точки, лежащей на радиусе, также для определения длины сторон в классе Side есть метод Length. Для удобства в классах Triangle и Quadrant, можно обратится и вернуть их точную площадь за счёт метода Square и за счёт метода CheckPoint, принимающего экземпляр класса Point, определить находится ли точка внутри фигуры. Диаграмма классов объектно-ориентированного приложения представлена на рис. 7.



**Program**

Класс



Методы



Main() : void

**Menu**

Класс



Поля



buf : double



buf\_p1 : Point



buf\_p2 : Point



g : double



index : int



menuItems : List<string>



p1 : Point



p2 : Point



Методы



CheckedReadLine() : double



Draw() : void



DrawMenu(List<string> items) : string



Menu()



PressAnyKeyToContinue() : void



SwapValues(ref double a, ref double b) : void

**MonteCarloCalculator**

Класс



Поля



d : Point



e : Point



g : Point



N : int



o : Point



p1 : Point



p2 : Point



quadrant : Quadrant



s : double



triangle : Triangle



Методы



DrawTable() : void



GetSquare() : double



MonteCarloCalculator(Point p1, Point p2, double g, int N)



MonteCarloSquare() : double

**Quadrant**

Класс



Поля



center : Point



p1 : Point



p2 : Point



Методы



CheckPoint(Point p) : bool



Quadrant()



Quadrant(Point center, Point p1)



Quadrant(Quadrant q)



Radius() : double



Square() : double

**Triangle**

Класс



Поля



a : Side



b : Side



c : Side



p1 : Point



p2 : Point



p3 : Point



Методы



CheckPoint(Point p) : bool



InitializeSides(Point p1, Point p2, Point p3) : void



Perimeter() : double



Square() : double



Triangle()



Triangle(Point p1, Point p2, Point p3)



Triangle(Triangle t)

**Side**

Класс



Поля



p1 : Point



p2 : Point



Методы



Length() : double



Side()



Side(Point p1, Point p2)



Side(Side s)

**Point**

Класс



Поля



x : double



y : double



Методы



Point()



Point(double x, double y)



Point(Point p)

Рис. 7. Диаграмма классов объектно-ориентированного приложения

## 5.3. Описание программы

Класс Point представляет собой точку, задаваемую двумя координатами x, y. Диаграмма класса представлена на рис. 8.



Рис. 8. Диаграмма класса Point

Класс Point содержит следующие поля:

public double x – координата *x* точки;

public double y – координата *y* точки.

Класс Point содержит следующие конструкторы:

public Point() – по умолчанию;

public Point(double x, double y) – инициализации;

public Point(Point p) – копии.

Класс Side представляет собой отрезок для инициализации которого требуются две точки, также класс содержит метод для нахождения своей длины. Диаграмма класса представлена на рис. 9.



Рис. 9. Диаграмма класса Side

Класс Side содержит следующие поля:

public Point p1 – первая точка;

public Point p2 – вторая точка.

Класс Side содержит следующие конструкторы:

public Side() – по умолчанию;

public Side(Point p1, Point p2) – инициализации;

public Side(Side s) – копии.

Класс Side содержит следующие методы:

public double Length() – считает длину стороны, используя координаты точек и теорему Пифагора.

Класс Triangle представляет собой треугольник, задаваемый тремя точками. Класс содержит методы по поиску точной площади, периметра, проверки наличия точки внутри треугольника, а также метод для инициализации сторон, используемый в конструкторах. Диаграмма класса представлена на рис. 10.



Рис. 10. Диаграмма класса Triangle

Класс Triangle содержит следующие поля:

Point p1, p2, p3 – вершины треугольника;

Point a, b, c – стороны треугольника.

Класс Triangle содержит следующие конструкторы:

public Triangle () – по умолчанию;

public Triangle (Point p1, Point p2, Point p3) – инициализации;

public Triangle (Triangle t) – копии.

Класс Triangle содержит следующие методы:

private void InitializeSides(Point p1, Point p2, Point p3) – инициализирует стороны треугольника a, b, c (используется в конструкторах);

public double Perimeter() – возвращает периметр треугольника;

public double Square()– возвращает точную площадь треугольника;

public bool CheckPoint(Point p) – проверяет наличие точки p внутри треугольника.

Класс Quadrant представляет собой четверть круга (квадрант), задаваемый двумя точками. Присутствуют методы по поиску точной площади, длины радиуса и проверки наличия точки внутри квадранта. Диаграмма класса представлена на рис. 11.

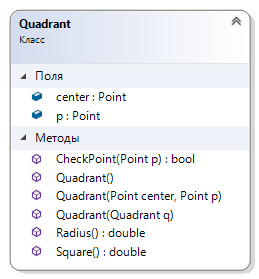


Рис. 11. Диаграмма класса Quadrant

Класс Quadrant содержит следующие поля:

Point center, p – центр и точка, лежащая на радиусе квадранта.

Класс Quadrant содержит следующие конструкторы:

public Quadrant () – по умолчанию;

public Quadrant (center, Point p1) – инициализации;

public Quadrant (Quadrant q) – копии.

Класс Quadrant содержит следующие методы:

public double Radius() – возвращает длину радиуса;

public double Square()– возвращает точную площадь квадранта;

public bool CheckPoint(Point p) – проверяет наличие точки p внутри квадранта.

Класс MonteCarloCalculator представляет собой объект для расчёта площади методом Монте-Карло, реальной площади, относительной погрешности, подсчёта времени за которое все эти вычисления были сделаны. Диаграмма класса представлена на рис. 12.

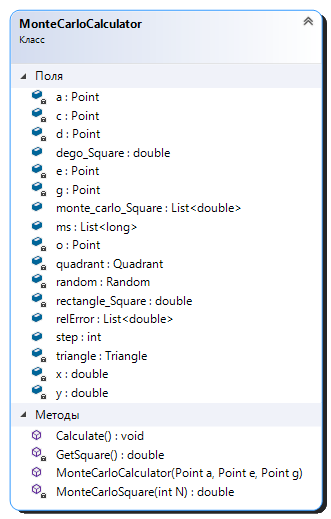


Рис. 12. Диаграмма класса MonteCarloCalculator

Класс MonteCarloCalculator содержит следующие поля:

Point d, e, g, o – точки нашей фигуры, с помощью которых мы создадим экземпляры классов Triangle и Quadrant, сами точки мы получим используя информацию переданную в класс инициализации;

Point a, c – точки, лежащие на главной диагонали прямоугольника в котором находится наша фигура;

Quadrant quadrant – квадрант, являющийся частью нашей фигуры и содержащий необходимые для наших расчётов методы;

Triangle triangle – треугольник, являющийся частью нашей фигуры и содержащий необходимые для наших расчётов методы;

public double rectangle\_Square – площадь прямоугольник;

double x,y - координаты для броска точки;

List<double> relError – список с значениями относительной погрешности;

public List<double> monte\_carlo\_Square – список с значениями площади фигуры, вычисленной с помощью метода Монте-Карло;

public double dego\_Square – точная площадь фигуры *dego*;

public List<long> ms – список со значениями затраченого времени на вычисления.

Класс MonteCarloCalculator содержит следующие конструкторы:

public MonteCarloCalculator(Point a, Point e, Point g) – конструктор инициализации, где входными данными будут точки a, e, g (точки уже были проверены в классе Menu).

Класс MonteCarloCalculator содержит следующие методы:

private double GetSquare() – возвращает точную площадь фигуры;

private double MonteCarloSquare() – метод отвечает за получение площади методом Монте-Карло;

public void Calculate() – метод заполняет все поля класса MonteCarloCalculator необходимые для вывода в таблицу с помощью класса Table.

Класс Table предназначен для вывода данных, полученных методом Calculate класса MonteCarloCalculator. Диаграмма класса представлена на рис. 13.

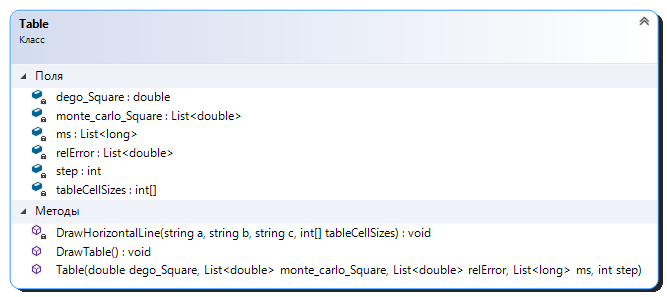


Рис. 13. Диаграмма класса Table

Класс Menu содержит следующие поля:

int[] tableCellSizes – массив с размерами ячеек таблицы;

double dego\_Square – точная площадь фигуры;

List<double> monte\_carlo\_Square – список, содержащий значения площади, полученной методом Монте-Карло;

List<double> relError – список, содержащий значения относительной погрешности;

List<long> ms = new List<long>() – список, содержащий время затраченное на выполнения алгоритма.

Класс Table содержит следующие конструкторы:

public Table(MonteCarloCalculator calc) – конструктор инициализации.

Класс Table содержит следующие методы:

public void DrawTable() – выводит данные таблицы в консоль.

private void DrawHorizontalLine(string a, string b, string c, int[] tableCellSizes) – метод , используется в DrawTable для вывода верхней и нижней линии таблицы;

Класс Menu представляет собой объект, обеспечивающий графический интерфейс в консоли для ввода и проверки данных, вводимых пользователем, для запуска вычислений, используя данные из заранее заготовленного примера или введенные пользователем. Для вычислений будет необходимо создание экземпляра класса MonteCarloCalculator, который будет осуществлять все расчёты и экземпляра класса Table для вывода полученных данных в консоль, также в меню имеется пункт для демонстрации текущих данных, которые будут использоваться для вычислений, а также пункт для выхода из программы. Диаграмма класса представлена на рис. 14.

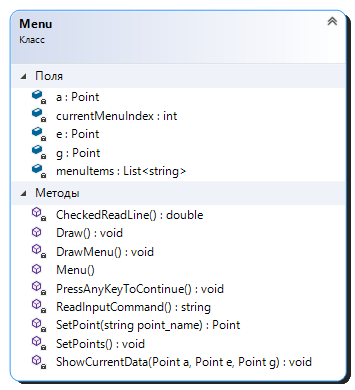


Рис. 14. Диаграмма класса Menu

Класс Menu содержит следующие поля:

private Point a, e, g – точки, являющиеся исходными данными для работы приложения;

private readonly List<string> menuItems – список пунктов меню;

private int currentMenuIndex – индекс, который определяет на каком пункте меню пользователь в данный момент находится.

Класс Menu содержит следующие конструкторы:

public Menu() – стандартный конструктор, где заполняются поля точек: *a, e ,g* и список пуктов меню.

Класс Menu содержит следующие методы:

private void PressAnyKeyToContinue() – вызывает сообщение “Press any key to continue…” и ставит программу в режим ожидания нажатия клавиши пользователем;

private static double CheckedReadLine() – метод отвечает за ввод с проверкой строки на то является ли её содержимое числом;

private static void ShowCurrentData(Point a, Point e, Point g) – метод выводящий в консоль в форматированном виде текущие данные приложения;

Point SetPoint(string point\_name) – метод для форматированной инициализации точки в консоли;

void SetPoints() – метод инициализирует точки a, e, g и с помощью экземпляра класса PointValidator проверяет точки на корректность и по этим точкам можно построить нашу фигуру, то на этом процесс инициализации окончится иначе PointValidator вернёт список ошибок и процесс инициализации начнётся заного;

void DrawMenu() – метод отвечает за то, как будет выглядеть меню;

string ReadInputCommand() – метод отвечает за управление меню и при нажатии клавиши “Enter” возвращает пункт меню, соответствующий индексу;

public void Draw() – представляет собой бесконечный цикл, который вызывает метод DrawMenu, switch case конструкцию принимающую строку, созвращаемую методом ReadInputCommand, а также соответственные алгоритмы действий в зависимости от выбранного пункта меню.

Класс PointValidator предназначен для проверка инициализации точек. Диаграмма класса представлена на рис. 15.

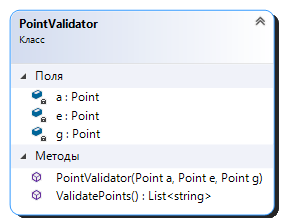


Рис. 15. Диаграмма класса Menu

Класс PointValidator содержит следующие поля:

private Point a, e, g – точки, которые необходимо проверить.

Класс Menu содержит следующие конструкторы:

public PointValidator(Point a, Point e, Point g) – инициализации, где заполняются поля точек: *a, e ,g*.

Класс PointValidator содержит следующие методы:

public List<string> ValidatePoints() – возвращает список ошибок инициализации, если такие пристутствуют.

Класс Program за счёт метода Main создаёт экземпляр класса Menu и вызывает метод этого класса Draw. Диаграмма класса представлена на рис. 16.



Рис. 16. Диаграмма класса Program

Класс Program не содержит полей.

Класс Program не содержит конструкторов.

Класс Program содержит следующие методы:

static void Main()- метод создаёт экземпляр класса Menu и вызывает метод этого класса Draw.

## 5.4. Руководство системного программиста

**5.4.1. Общие сведения о приложении**

Приложение разработано для вычисления площади фигуры методом Монте-Карло. Данное приложение реализовано как консольное на языке программирования C# с использованием объектно-ориентированного подхода.

**5.4.2. Системные требования**

Для корректной работы приложения требуется:

* Операционная система Windows 7 и выше;
* платформа .NET Core 3.1 и выше;
* архитектура 32-разрядная (х86)/64-разрядная (х64);
* оперативная память 512 МБ и более;
* устройства ввода: клавиатура и мышь.

**5.4.3. Структура программы**

Программа состоит из классов: Program, Menu, MonteCarloCalculator, Triangle, Quadrant, Side, Point. Класс Program содержит главный метод Main, который создаёт экземпляр класса Menu, который обеспечивает графический интерфейс для нашего приложения. С помощью пунктов меню «Set Input Data» выполняется заполнение полей, необходимых для создания экземпляров класса MonteCarloCalculator и последующего вызова методов Calculate и DrawTable, необходимых для расчёта данных и вывода их в консоль в виде таблицы. Создание и вызов происходит за счёт пункта меню «Calculate», а также пункта «Show an Example», который использует данные из заранее заготовленного примера. Класс MonteCarloCalculator использует такие, классы как: Triangle, Quadrant, Side, Point: их экземпляры и методы для произведения расчётов.

**5.4.4. Проверка программы**

Для тестирования программы можно выбрать расчёт, используя заранее подготовленные исходные данные. В результатах вычислений площадь фигуры должна составлять 44,625 единиц.

**5.4.5. Сообщения системному программисту**

В ходе выполнения программы возможен вывод следующих сообщений об ошибках:

* Error: AG length equal 0 or point G below point A;
* Error: AB length equal 0 or point E below point A;
* Error: AO length equal 0 or point E is to the left of point A;
* Error: AG are not parallel to the axis Y;
* Error: AG length more than length AB.

При появлении данных ошибок программа предложит повторно ввести координаты точек x и y указанных, как исходные данные. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* You didn't set data.

Эта ошибка возникает при выборе пункта «Calculate» без введённых данных. Далее вас вернёт в изначальное меню. Для того, чтобы её не возникало необходимо воспользоваться пунктом меню «Set Input Data».

* Incorrect data format, please enter data again:

Данная ошибка возникает, если в поле для ввода информации был введён неверный формат данных, после вывода данного сообщения пользователю необходимо ввести данные ещё раз.

В ходе выполнения программы возможен вывод следующих сообщений:

* Press any key to continue...

Данное сообщение возникает каждый раз, когда программа приостановлена и ожидает нажатия любой клавиши.

* Please, enter correct data

После данного сообщения программа перенесёт вас обратно к инициализации точек.

## 5.5. Руководство программиста

**5.5.1. Назначение и условие применения программы**

Приложение разработано для вычисления площади фигуры методом Монте-Карло. Программа обладает меню (навигация по меню осуществляется с помощью стрелочек) в котором содержатся следующие пункты:

* «Set Input Data» - самостоятельный ввод данных;
* «Show an Example» - произведение расчётов и вывод таблицы исходя из данных заранее заготовленного примера;
* «Show Current Data» - для просмотра текущих данных;
* «Calculate» - для запуска расчётов и вывода таблицы с данными в консоль;
* «Exit» - выход из программы, также для выхода можно воспользоваться клавишей «Escape»;
* Для корректной работы приложения требуется:
* Операционная система Windows 7 и выше;
* платформа .NET Core 3.1 и выше;
* архитектура 32-разрядная (х86)/64-разрядная (х64);
* оперативная память 512 МБ и более;
* устройства ввода: клавиатура и мышь.

**5.5.2 Характеристики** **программы**

За вычетом времени, затрачиваемого пользователем на ввод координат и выбор команд, программа выполняется за 1-3 секунды.

**5.5.3 Входные и выходные данные**

Вся входная информация (координаты вершин и команды) получается программой из консоли. Вся выходная информация (сообщения и результаты) выводится программой в консоль.

**5.5.4 Сообщения**

В ходе выполнения программы возможен вывод следующих сообщений об ошибках:

* Error: AG length equal 0 or point G below point A;
* Error: AB length equal 0 or point E below point A;
* Error: AO length equal 0 or point E is to the left of point A;
* Error: AG are not parallel to the axis Y;
* Error: AG length more than length AB.

При появлении данных ошибок программа предложит повторно ввести координаты точек x и y указанных, как исходные данные. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* You didn't set data

Эта ошибка возникает при выборе пункта «Calculate» без введённых данных. Далее вас вернёт в изначальное меню. Для того, чтобы её не возникало необходимо воспользоваться пунктом меню «Set Input Data».

* Incorrect data format, please enter data again:

Данная ошибка возникает, если в поле для ввода информации был введён неверный формат данных, после вывода данного сообщения пользователю необходимо ввести данные ещё раз.

В ходе выполнения программы возможен вывод следующих сообщений:

* Press any key to continue...

Данное сообщение возникает каждый раз, когда программа приостановлена и ожидает нажатия любой клавиши.

* Please, enter correct data

После данного сообщения программа перенесёт вас обратно к инициализации точек.

## 5.6. Руководство оператора

**5.6.1 Назначение программы**

Приложение разработано для вычисления площади фигуры методом Монте-Карло. Программа обладает меню (навигация по меню осуществляется с помощью стрелочек) в котором содержатся следующие пункты:

* «Set Input Data» - самостоятельный ввод данных;
* «Show an Example» - произведение расчётов и вывод таблицы исходя из данных заранее заготовленного примера;
* «Show Current Data» - для просмотра текущих данных;
* «Calculate» - для запуска расчётов и вывода таблицы с данными в консоль;
* «Exit» - выход из программы, также для выхода можно воспользоваться клавишей «Escape»;

**5.6.2. Условия выполнения программы**

Для корректной работы приложения требуется:

* Операционная система Windows 7 и выше;
* платформа .NET Core 3.1 и выше;
* архитектура 32-разрядная (х86)/64-разрядная (х64);
* оперативная память 512 МБ и более;
* устройства ввода: клавиатура и мышь.

**5.6.3. Выполнение программы**

Приложение разработано для вычисления площади фигуры методом Монте-Карло. Программа обладает меню (навигация по меню осуществляется с помощью стрелочек) в котором содержатся следующие пункты:

* «Set Input Data» - самостоятельный ввод данных;
* «Show an Example» - произведение расчётов и вывод таблицы исходя из данных заранее заготовленного примера;
* «Show Current Data» - для просмотра текущих данных;
* «Calculate» - для запуска расчётов и вывода таблицы с данными в консоль;
* «Exit» - выход из программы, также для выхода можно воспользоваться клавишей «Escape»;

**5.6.4. Сообщения оператору**

В ходе выполнения программы возможен вывод следующих сообщений об ошибках:

* Error: AG length equal 0 or point G below point A;
* Error: AB length equal 0 or point E below point A;
* Error: AO length equal 0 or point E is to the left of point A;
* Error: AG are not parallel to the axis Y;
* Error: AG length more than length AB.

При появлении данных ошибок программа предложит повторно ввести координаты точек x и y указанных, как исходные данные. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* You didn't set data

Эта ошибка возникает при выборе пункта «Calculate» без введённых данных. Далее вас вернёт в изначальное меню. Для того, чтобы её не возникало необходимо воспользоваться пунктом меню «Set Input Data».

* Incorrect data format, please enter data again:

Данная ошибка возникает, если в поле для ввода информации был введён неверный формат данных, после вывода данного сообщения пользователю необходимо ввести данные ещё раз.

В ходе выполнения программы возможен вывод следующих сообщений:

* Press any key to continue...

Данное сообщение возникает каждый раз, когда программа приостановлена и ожидает нажатия любой клавиши.

* Please, enter correct data

После данного сообщения программа перенесёт вас обратно к инициализации точек.

## 5.7. Результаты работы

Результаты работы приложения представлены на рис. 17.

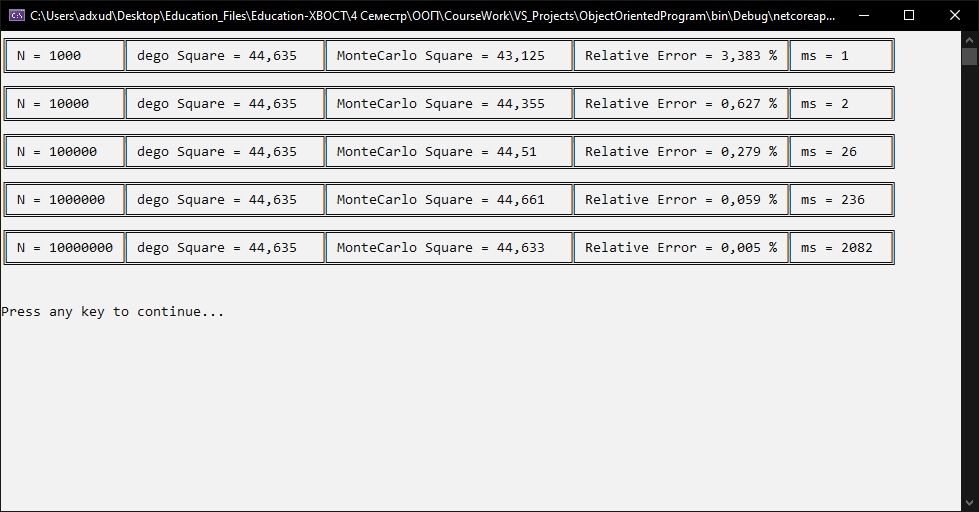


Рис. 17. Тестирование объектно-ориентированного приложения

# АНАЛИЗ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ ПРИЛОЖЕНИЙ

Приложения тестировались на компьютере со следующими характеристиками:

* Операционная система Windows 10;
* Процессор Intel(R) Core (TM) i3-6100 с частотой 3.70 GHz;
* Оперативная память 12 ГБ;
* Видеокарта AMD Radeon R7 370.

Результаты сравнения двух приложений представлены в таблице 1.

Таблица 1 – продолжительность вычислений в двух приложениях

|  |  |  |
| --- | --- | --- |
| Количество точек | Время выполнения, мс | |
| Процедурное приложение на базе Console/C# | Объектно-ориентированное приложение на базе Console/C# |
|  | 1 | 1 |
|  | 1 | 2 |
|  | 18 | 26 |
|  | 159 | 236 |
|  | 1578 | 2082 |

Процедурное приложение работает несколько быстрее объектно-ориентированного.

# УЛУЧШЕНИЕ ХАРАКТЕРИСТИК КАЧЕСТВА ПРИЛОЖЕНИЙ

## 7.1. Использование встроенной оптимизации кода

После завершения написания приложения необходимо его оптимизировать. Для этого сначала нужно в свойствах проекта (рис. 17) поменять конфигурацию с Debug на Release. Это увеличит производительность приложения из-за того, что в режиме Debug отключены все настройки по оптимизации и часть ресурсов уходи на сбор и хранение информации об отладке.

Также в свойствах проекта во вкладке «Сборка» необходимо отметить пункт «Оптимизировать код» (рис. 17), что сильно скажется на производительности программы.

В этом же окне нужно снять галочку с пункта «Предпочтительно 32-разр.» (рис. 17). Это серьёзно увеличит производительность, поскольку приложение тестируется на 64-разрядной системе.

В приложении никак не используется класс Trace, поэтому можно снять галочку с пункта «Определить константу TRACE» (рис. 18), что в итоге незначительно увеличит производительность.

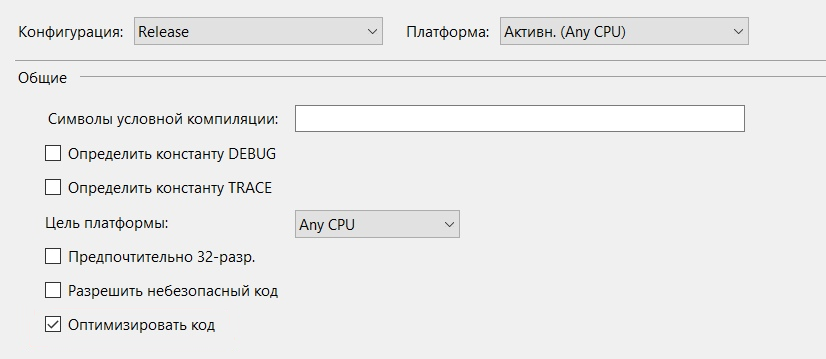


Рис. 18. Вкладка «Сборка» в свойствах проекта

## 7.2. Профилирование

Для приложений было проведено профилирование по использованию ЦП. Результаты приведены на рис. 19 и 20.

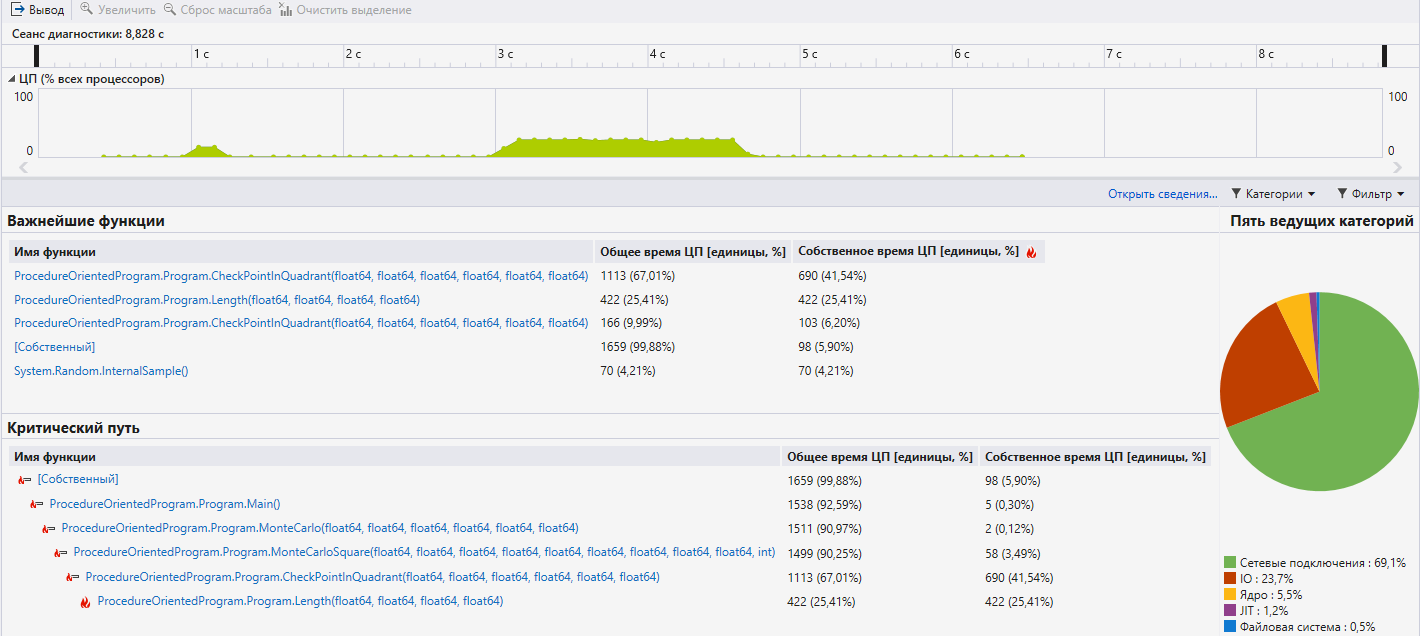


Рис. 19. Результаты профилирования процедурного приложения

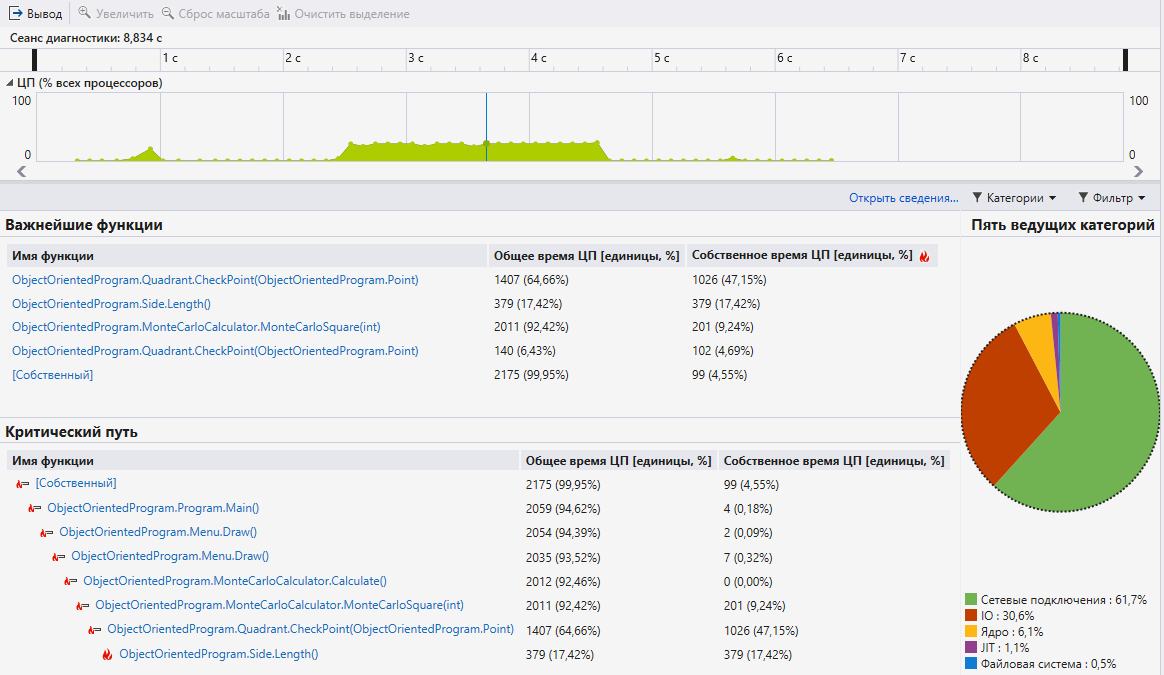


Рис. 20. Результаты профилирования объектно-ориентированного приложения

Из результатов видно, что больше всего ресурсов уходит на методы Menu.Draw(), MonteCarloCalculator.Calculate() и MonteCarloCalculator.MonteCarloSquare() в объектно-ориентированном приложении, а в процедурном метод MonteCarlo(). Объектно-ориентированное приложение нагружает систему чуть больше, чем процедурно-ориентированное.

## 7.3. Результаты улучшения характеристик

После оптимизации кода производительность приложений сильно возросла. Сравнение результатов до и после оптимизации представлено в таблице 2.

Таблица 2 – сравнение приложений до и после оптимизации

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество точек | Время выполнения, мс | | | |
| Процедурное приложение на базе Console/C# | | Объектно-ориентированное приложение на базе Console/C# | |
| До оптим. | После оптим. | До оптим. | После оптим. |
|  | 1 | 1 | 1 | 1 |
|  | 1 | 1 | 2 | 1 |
|  | 18 | 13 | 26 | 19 |
|  | 159 | 133 | 236 | 188 |
|  | 1578 | 1312 | 2082 | 1593 |

В среднем процедурное приложение стало работать на 17% быстрее, а объектно-ориентированное приложение на 23% быстрее.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были выполнены все основные цели и задачи, а именно: закреплены знания по курсу "Объектно-ориентированное программирование" и приобретены навыки объектно-ориентированной и процедурной реализаций прикладной задачи (задачи вычисления площади геометрической фигуры методом Монте Карло) с использованием различных языков, инструментальных систем и библиотек, автоматизирующих проектирование, программирование и отладку создаваемых приложений.

Также были приобретены навыки решения вычислительных задач; практически освоены современные инструментальные системы разработки ПО; был проведён сравнительный анализ вычислительной эффективности процедурных и объектно-ориентированных программ; получены навыки создания программ на языках C++ и C# платформы Microsoft .NET Core; приобретены практические навыки оформления и выпуска документации в соответствии с требованиями стандартов (ЕСПД, UML).

# СПИСОК ЛИТЕРАТУРЫ

1. Microsoft Docs [Электронный ресурс] : Документация по C#. Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/csharp/ (дата обращения 20.05.2020).
2. Metanit [Электронный ресурс] : Полное руководство по языку программирования С# 8.0 и платформе .NET Core 3.1. Режим доступа: https://metanit.com/sharp/tutorial/ (дата обращения 21.05.2020).
3. Wikipedia [Электронный ресурс] : Метод Монте-Карло. Режим доступа: https://ru.wikipedia.org/wiki/Метод\_Монте-Карло (дата обращения 22.04.2020).
4. Wikipedia [Электронный ресурс] : UML. Режим доступа: https://ru.wikipedia.org/wiki/ UML (дата обращения 23.04.2020).
5. Троелсен Э., Джепикс Ф. Язык программирования C# 7 и платформы. NET и. NET Core. – Litres, 2019.
6. Metanit [Электронный ресурс] : Учебник по языку С# 7.0 и платформе .NET 4.7. Режим доступа:  [https://metanit.com/sharp/](https://vk.com/away.php?utf=1&to=https%3A%2F%2Fmetanit.com%2Fsharp%2F) (дата обращения 25.05.2020).
7. Буч Г. Язык UML: Руководство пользователя / Г.Буч, Д.Рамбо, И.Якобсон; пер.с англ.Мухин Н. — 2-е изд. — М. : ДМК Пресс:Академия Айти, 2007 .— 496с.

# ПРИЛОЖЕНИЕ

## Файл Program.cs (процедурное приложение)

**using** System;

**using** System.Collections.Generic;

**using** System.Diagnostics;

**namespace** ProcedureOrientedProgram {

**class** Program {

/\* В методе Main вызываются методы DrawMenu и ReadInputData, обеспечивающие работу меню,

         \* а также алгоритмы действий при выборе определенных пунктов меню\*/

**static** **void** Main() {

// Введение полей, необходимых для описания фигуры

**double** ax = 0;

**double** ay = 0;

**double** ex = 0;

**double** ey = 0;

**double** gx = 0;

**double** gy = 0;

/\* Объявление цикла, обновляющий меню при нажатии клавиш управления,

             \* а также содержащий алгоритмы действий при выборе того или иного пункта меню\*/

**while** (**true**) {

DrawMenu();

//Switch, содержащий алгоритмы действий

**switch** (ReadInputCommand()) {

// 1.

// Пункт меню, запускающий ввод данных вручную

**case** "Set Input Data":

Console.Clear();

// Метод для инициализации координат точек,содержащий проверку на ошибки.

SetPoints(**ref** ax, **ref** ay, **ref** ex, **ref** ey, **ref** gx, **ref** gy);

ShowCurrentData(ax, ay, ex, ey, gx, gy);

**break**;

// 2.

/\* Пункт меню с помощью которого мы можем посмотреть, как работает программа,

                     \* используя готовый пример исходных данных\*/

**case** "Show an Example":

// Задаются исходные данные

ax = 0;

ay = 0;

ex = 10;

ey = 5;

gx = 0;

gy = 3;

ShowCurrentData(ax, ay, ex, ey, gx, gy);

// Вызов нашего основного метода MonteCarlo, используюя пример данных

MonteCarlo(ax, ay, ex, ey, gx, gy);

PressAnyKeyToContinue();

**break**;

// 3.

// Пункт меню, выводящий в консоль текущие данные нашей фигуры

**case** "Show Current Data":

ShowCurrentData(ax, ay, ex, ey, gx, gy);

**break**;

// 4.

// Пункт меню запускающий вычисления

**case** "Calculate":

Console.Clear();

// Проверка на то были ли введены данные

**if** (ax == ex) {

Console.WriteLine("You don't set data");

PressAnyKeyToContinue();

} **else** {

/\* Вызов нашего основного метода MonteCarlo,

                             \* используюя ранее введенные пользователем данные\*/

MonteCarlo(ax, ay, ex, ey, gx, gy);

PressAnyKeyToContinue();

}

**break**;

// 5.

// Пункт меню, обеспечивающий выход из программы

**case** "Exit":

Environment.Exit(0);

**break**;

}

}

}

// Метод MonteCarlo, осуществляющего расчет данных и вывод их в таблицу

**public** **static** **void** MonteCarlo(**double** ax, **double** ay, **double** ex, **double** ey, **double** gx, **double** gy) {

// Площадь прямоугольника в котором находятся фигуры

**double** rectangle\_Square = (ex + (ey - ay) - ax) \* (ey - ay);

// Задаем координаты недостаяющих точек dego

**double** ox = ex;

**double** oy = ay;

**double** dx = ex + (ey - ay);

**double** dy = ay;

// Cписок в который будет записываться площадь, вычисленная методом Монте-Карло

List<**double**> monte\_carlo\_Square = **new** List<**double**>();

/\* Cписок в который будет записываться относительная ошибка

             \* вычислений каждой итерации метода Монте-Карло\*/

List<**double**> relError = **new** List<**double**>();

// Cписок в который будет записываться время затраченное на вычисления

List<**long**> ms = **new** List<**long**>();

// Поле, которе будет отвечать за количествво итераций цикла, описанного ниже

**int** step = 0;

// Поле в которое будет записываться точная площадь фигуры

**double** dego\_Square = 0;

**for** (**int** N = Convert.ToInt32(Math.Pow(10, 3)); N <= Math.Pow(10, 7); N \*= 10) {

// Объявляем таймер

Stopwatch sw = **new** Stopwatch();

// Старт таймера

sw.Start();

dego\_Square = GetSquare(dx, dy, ex, ey, gx, gy, ox, oy);

monte\_carlo\_Square.Add(MonteCarloSquare(dx, dy, ex, ey, gx, gy, ox, oy, rectangle\_Square, N));

relError.Add(Math.Abs((dego\_Square - monte\_carlo\_Square[step]) / dego\_Square) \* 100);

// Стоп таймера

sw.Stop();

// Подсчет милисекунд

ms.Add(sw.ElapsedMilliseconds);

step++;

}

// Вывод таблицы в консоль

DrawTable(step, dego\_Square, monte\_carlo\_Square, relError, ms);

}

// Метод возвращающий точную площадь фигуры

**static** **double** GetSquare(**double** dx, **double** dy, **double** ex, **double** ey, **double** gx, **double** gy, **double** ox, **double** oy) {

**return** TriangleSquare(ex,ey,gx,gy,ox,oy) + QuadrantSquare(ox, oy, dx, dy);

}

// Площадь фигуры методом Монте-Карло

**static** **double** MonteCarloSquare(**double** dx, **double** dy, **double** ex, **double** ey, **double** gx, **double** gy, **double** ox, **double** oy, **double** rectangle\_Square, **int** N) {

Random random = **new** Random();

**int** count = 0;

**for** (**int** i = 0; i < N; i++) {

**double** x = random.NextDouble() \* (dx-gx) + gx;

**double** y = random.NextDouble() \* (ey-dy) + dy;

**if** (CheckPointInTriangle(x, y, ex,ey,gx,gy,ox,oy) || CheckPointInQuadrant(x, y, ox, oy, dx, dy)) {count++;}

}

**return** rectangle\_Square \* count / N;

}

// Проверка наличия точки в треугольнике

**static** **bool** CheckPointInTriangle(**double** x0, **double** y0, **double** x1, **double** y1, **double** x2, **double** y2, **double** x3, **double** y3) {

**double** buf1 = (x1 - x0) \* (y2 - y1) - (x2 - x1) \* (y1 - y0);

**double** buf2 = (x2 - x0) \* (y3 - y2) - (x3 - x2) \* (y2 - y0);

**double** buf3 = (x3 - x0) \* (y1 - y3) - (x1 - x3) \* (y3 - y0);

**if** ((buf1 >= 0 && buf2 >= 0 && buf3 >= 0) || (buf1 <= 0 && buf2 <= 0 && buf3 <= 0)) {

**return** **true**;

} **else** **return** **false**;

}

// Проверка наличия точки в квадранте

**static** **bool** CheckPointInQuadrant(**double** x, **double** y, **double** center\_of\_circle\_x, **double** center\_of\_circle\_y, **double** point\_on\_radius\_x, **double** point\_on\_radius\_y) {

**double** R = Length(center\_of\_circle\_x, center\_of\_circle\_y, point\_on\_radius\_x, point\_on\_radius\_y);

**if** ((Math.Pow(x - center\_of\_circle\_x, 2) + Math.Pow(y - center\_of\_circle\_y, 2) <= Math.Pow(R, 2)) && (x >= center\_of\_circle\_x) && (y >= center\_of\_circle\_y)) {

**return** **true**;

} **else** **return** **false**;

}

// Вычисление точной площади квадранта

**static** **double** QuadrantSquare(**double** x1, **double** y1, **double** x2, **double** y2) {

**return** Math.Pow(Length(x1, y1, x2, y2), 2) \* 0.25 \* Math.PI;

}

// Вычисление точной площади треугольника

**static** **double** TriangleSquare(**double** x1, **double** y1, **double** x2, **double** y2, **double** x3, **double** y3) {

**double** p = (Length(x1, y1, x2, y2) + Length(x2, y2, x3, y3) + Length(x3, y3, x1, y1)) / 2;

**return** Math.Sqrt(p \* (p - Length(x1, y1, x2, y2)) \* (p - Length(x2, y2, x3, y3)) \* (p - Length(x3, y3, x1, y1)));

}

// Вычисление длины отрезка

**static** **double** Length(**double** x1, **double** y1, **double** x2, **double** y2) {

**return** Math.Sqrt(Math.Pow(x1 - x2, 2) + Math.Pow(y1 - y2, 2));

}

// Методы для таблицы

// 2 метода для вывода таблицы в консоль

**static** **void** DrawTable(**int** step, **double** dego\_Square, List<**double**> monte\_carlo\_Square, List<**double**> relError, List<**long**> ms) {

**int**[] tableCellSizes = **new** **int**[] { 14, 24, 30, 26, 12 };

**for** (**int** i = 0; i < step; i++) {

DrawHorizontalLine("╔", "╦", "╗", tableCellSizes);

Console.WriteLine("║ N = {0,-8} ║ dego Square = {1,-8} ║ MonteCarlo Square = {2,-8} ║ Relative Error = {3,-6}% ║ ms = {4,-5} ║", Math.Pow(10, 3 + Convert.ToDouble(i)), Math.Round(dego\_Square, 3), Math.Round(monte\_carlo\_Square[i], 3), Math.Round(relError[i], 3), ms[i]);

DrawHorizontalLine("╚", "╩", "╝", tableCellSizes);

}

}

**static** **void** DrawHorizontalLine(**string** a, **string** b, **string** c, **int**[] tableCellSizes) {

Console.Write(a);

**for** (**int** j = 0; j < tableCellSizes.Length; j++) {

**for** (**int** i = 0; i < tableCellSizes[j]; i++) {

Console.Write("═");

}

**if** (j != tableCellSizes.Length - 1) {

Console.Write(b);

}

}

Console.WriteLine(c);

}

// Метод для инициализации точек

**static** **void** SetPoints(**ref** **double** ax, **ref** **double** ay, **ref** **double** ex, **ref** **double** ey, **ref** **double** gx, **ref** **double** gy) {

SetPoint("a", **ref** ax, **ref** ay);

SetPoint("e", **ref** ex, **ref** ey);

SetPoint("g", **ref** gx, **ref** gy);

List<**string**> error\_List = ErrorList(ax, ay, ex, ey, gx, gy);

**if** (error\_List.Count == 0)

**return**;

Console.WriteLine("Number of errors = " + error\_List.Count + ";\n");

**foreach** (**string** error **in** error\_List)

Console.WriteLine(error);

Console.WriteLine("\nPlease enter correct data");

PressAnyKeyToContinue();

SetPoints(**ref** ax, **ref** ay, **ref** ex, **ref** ey, **ref** gx, **ref** gy);

}

// Метод для инициализации точки

**static** **void** SetPoint(**string** point\_name, **ref** **double** x, **ref** **double** y) {

Console.WriteLine("Set coordinates for point '" + point\_name + "'\n");

Console.Write("\nSet x:");

x = CheckedReadLine();

Console.Write("\nSet y:");

y = CheckedReadLine();

Console.Clear();

}

// Метод возвращающий список ошибок

**static** List<**string**> ErrorList(**double** ax, **double** ay, **double** ex, **double** ey, **double** gx, **double** gy) {

List<**string**> errorMessages = **new** List<**string**>();

**if** (gy - ay <= 0) {

errorMessages.Add("Error: AG length equal 0 or point G below point A");

}

**if** (ey - ay <= 0) {

errorMessages.Add("Error: AB length equal 0 or point E below point A");

}

**if** (ex - ax <= 0) {

errorMessages.Add("Error: AO length equal 0 or point E is to the left of point A");

}

**if** (ax != gx) {

errorMessages.Add("Error: AG are not parallel to the axis Y");

}

**if** (Math.Abs(gy - ay) >= Math.Abs(ey - ay)) {

errorMessages.Add("Error: AG length more than length AB");

}

**return** errorMessages;

}

// Метод, проверяющий является ли строка числом

**static** **double** CheckedReadLine() {

**if** (**double**.TryParse(Console.ReadLine(), **out** **double** digit)) {

**return** digit;

} **else** {

Console.Write("Incorrect data format, please enter data again: ");

**return** CheckedReadLine();

}

}

// Метод показывающий текущие данные фигуры

**static** **void** ShowCurrentData(**double** ax, **double** ay, **double** ex, **double** ey, **double** gx, **double** gy) {

Console.Clear();

Console.WriteLine("Your figure data:\n\nd({0}, {1}), e({2}, {3}), g({4}, {5}), o({6}, {7});", ex + (ey - ay), ay, ex, ey, gx, gy, ex, ay);

PressAnyKeyToContinue();

}

// Метод сообщающий, что нужно нажать на клавишу для того, чтобы продолжить

**static** **void** PressAnyKeyToContinue() {

Console.WriteLine("\n\nPress any key to continue...");

Console.ReadKey();

Console.Clear();

}

// Создание списка пунктов меню

**static** List<**string**> menuItems = **new** List<**string**>()

{

"Set Input Data",

"Show an Example",

"Show Current Data",

"Calculate",

"Exit"

};

// Индекс пункта меню на котором находится пользователь

**static** **int** currentMenuIndex = 0;

// Метод, отвечающий за то, как выглядит меню

**static** **void** DrawMenu() {

Console.CursorVisible = **false**;

**for** (**int** i = 0; i < menuItems.Count; i++) {

**if** (i == currentMenuIndex) {

Console.BackgroundColor = ConsoleColor.Gray;

Console.ForegroundColor = ConsoleColor.Black;

Console.WriteLine(menuItems[i]);

} **else** {

Console.WriteLine(menuItems[i]);

}

Console.ResetColor();

}

}

// Метод, отвечающий за управление меню

**static** **string** ReadInputCommand() {

ConsoleKeyInfo ckey = Console.ReadKey();

**if** (ckey.Key == ConsoleKey.DownArrow && currentMenuIndex < menuItems.Count - 1) {

currentMenuIndex++;

} **else** **if** (ckey.Key == ConsoleKey.UpArrow && currentMenuIndex > 0) {

currentMenuIndex--;

} **else** **if** (ckey.Key == ConsoleKey.Enter) {

Console.Clear();

**return** menuItems[currentMenuIndex];

} **else** **if** (ckey.Key == ConsoleKey.Escape) {

Environment.Exit(0);

}

Console.Clear();

**return** "";

}

}

}

## Файл Program.cs (объектно-ориентированное приложение)

**namespace** ObjectOrientedProgram

{

**class** Program

{

**static** **void** Main()

{

**new** Menu().Draw();

}

}

}

## Файл Menu.cs

**using** System;

**using** System.Collections.Generic;

**namespace** ObjectOrientedProgram {

**class** Menu {

// Поле которое будет заполнено списком пунктов меню

**readonly** List<**string**> menuItems;

**private** **int** currentMenuIndex = 0;

/\* Введение полей, необходимых для описания фигуры и буферных переменных,

         \* которые будут использоваться для предотвращения ошибок при вводе данных\*/

Point a;

Point e;

Point g;

// Конструктор по умолчанию в котором мы заполняем список пунтов меню

**public** Menu() {

a = **new** Point();

e = **new** Point();

g = **new** Point();

menuItems = **new** List<**string**>()

{

"Set Input Data",

"Show an Example",

"Show Current Data",

"Calculate",

"Exit"

};

}

/\* Метод запускающий цикл, обновляющий меню при нажатии клавиш управления,

         \* а также содержащий алгоритмы действий при выборе того или иного пункта меню\*/

**public** **void** Draw() {

// Объявление цикла

// Switch, содержащий алгоритмы действий

DrawMenu();

**switch** (ReadInputCommand()) {

// 1.

// Пункт меню, запускающий ввод данных вручную

**case** "Set Input Data":

//Console.Clear();

SetPoints();

// Заполнение 'чистовых' полей

ShowCurrentData(a, e, g);

**break**;

// 2.

/\* Пункт меню с помощью которого мы можем посмотреть, как работает программа,

                 \* используя готовый пример исходных данных\*/

**case** "Show an Example":

// Задаются исходные данные

a = **new** Point(0, 0);

e = **new** Point(10, 5);

g = **new** Point(0, 3);

ShowCurrentData(a, e, g);

MonteCarloCalculator calc\_example = **new** MonteCarloCalculator(a, e, g);

calc\_example.Calculate();

Table table\_example = **new** Table(calc\_example);

table\_example.DrawTable();

PressAnyKeyToContinue();

**break**;

// 3.

// Пункт меню, выводящий в консоль текущие данные нашей фигуры

**case** "Show Current Data":

ShowCurrentData(a, e, g);

**break**;

// 4.

// Пункт меню запускающий вычисления

**case** "Calculate":

Console.Clear();

// Проверка на то были ли введены данные

**if** (a.x == e.x) {

Console.WriteLine("You didn't set the data");

PressAnyKeyToContinue();

}

**else**

{

MonteCarloCalculator calc = **new** MonteCarloCalculator(a, e, g);

calc.Calculate();

Table table = **new** Table(calc);

table.DrawTable();

PressAnyKeyToContinue();

}

**break**;

// 5.

// Пункт меню, обеспечивающий выход из программы

**case** "Exit":

Environment.Exit(0);

**break**;

**default**:

**break**;

}

Draw();

}

// Метод для инициализации координат точек a,e,g

**void** SetPoints() {

a = SetPoint("a");

e = SetPoint("e");

g = SetPoint("g");

// Проверка на ошибки

// Метод для создания сообщения об ошибке

/\* Метод включающий все возможные ошибки инициализации точек и если ошибки присутствуют,

            \* то метод снова вызывает метод для инициализации координат точек\*/

List<**string**> errorList = **new** PointValidator(a, e, g).ValidatePoints();

**if** (errorList.Count == 0)

**return**;

Console.WriteLine("Number of errors = " + errorList.Count + ";\n");

**foreach** (**var** error **in** errorList)

Console.WriteLine(error);

Console.WriteLine("\nPlease enter correct data");

PressAnyKeyToContinue();

SetPoints();

}

// Метод для инициализации точки

Point SetPoint(**string** point\_name) {

**var** point = **new** Point();

Console.WriteLine("Set coordinates for point '" + point\_name + "'\n");

Console.Write("\nSet x:");

point.x = CheckedReadLine();

Console.Write("\nSet y:");

point.y = CheckedReadLine();

Console.Clear();

**return** point;

}

// Метод, проверяющий является ли строка числом

**double** CheckedReadLine() {

**if** (**double**.TryParse(Console.ReadLine(), **out** **double** digit))

**return** digit;

Console.Write("Incorrect data format, please enter data again: ");

**return** CheckedReadLine();

}

// Метод показывающий текущие данные фигуры

**void** ShowCurrentData(Point a, Point e, Point g) {

Console.Clear();

Console.WriteLine("Your figure data:\n\nd({0}, {1}), e({2}, {3}), g({4}, {5}), o({6}, {7});", e.x + (e.y - a.y), a.y, e.x, e.y, g.x, g.y, e.x, a.y);

PressAnyKeyToContinue();

}

// Метод сообщающий, что нужно нажать на клавишу для того, чтобы продолжить

**void** PressAnyKeyToContinue() {

Console.WriteLine("\n\nPress any key to continue...");

Console.ReadKey();

Console.Clear();

}

/\* Метод DrawMenu.

        \* Он принимает список пунктов меню и в зависимости от текущего значения index при нажатии enter возвращает строку выбранного пункта меню.

        \* А с помощью стрелочек вверх и вниз он увеличивает или уменьшает значение переменной index в пределах, от 0 до размера, переданного

        \* в метод списка -1, также метод выделяет пункт меню, соответствующий по номеру текущему значению index.

        \*/

**void** DrawMenu() {

Console.CursorVisible = **false**;

**for** (**int** i = 0; i < menuItems.Count; i++) {

**if** (i == currentMenuIndex) {

Console.BackgroundColor = ConsoleColor.Gray;

Console.ForegroundColor = ConsoleColor.Black;

Console.WriteLine(menuItems[i]);

} **else** {

Console.WriteLine(menuItems[i]);

}

Console.ResetColor();

}

}

**string** ReadInputCommand() {

ConsoleKeyInfo ckey = Console.ReadKey();

**if** (ckey.Key == ConsoleKey.DownArrow && currentMenuIndex < menuItems.Count - 1) {

currentMenuIndex++;

} **else** **if** (ckey.Key == ConsoleKey.UpArrow && currentMenuIndex > 0) {

currentMenuIndex--;

} **else** **if** (ckey.Key == ConsoleKey.Enter) {

Console.Clear();

**return** menuItems[currentMenuIndex];

} **else** **if** (ckey.Key == ConsoleKey.Escape) {

Environment.Exit(0);

}

Console.Clear();

**return** "";

}

}

}

## Файл PointValidator.cs

**using** System;

**using** System.Collections.Generic;

**namespace** ObjectOrientedProgram {

**class** PointValidator {

Point a, e, g;

**public** PointValidator(Point a, Point e, Point g) {

**this**.a = a;

**this**.e = e;

**this**.g = g;

}

**public** List<**string**> ValidatePoints() {

List<**string**> error\_list = **new** List<**string**>();

**if** (g.y - a.y <= 0)

error\_list.Add("Error: AG length equal 0 or point G below point A;");

**if** (e.y - a.y <= 0)

error\_list.Add("Error: AB length equal 0 or point E below point A");

**if** (e.x - a.x <= 0)

error\_list.Add("Error: AO length equal 0 or point E is to the left of point A");

**if** (a.x != g.x)

error\_list.Add("Error: AG are not parallel to the axis Y");

**if** (Math.Abs(g.y - a.y) >= Math.Abs(e.y - a.y))

error\_list.Add("Error: AG length more than length AB");

**return** error\_list;

}

}

}

## Файл MonteCarloCalculator.cs

**using** System;

**using** System.Diagnostics;

**using** System.Collections.Generic;

**namespace** ObjectOrientedProgram {

**public** **class** MonteCarloCalculator {

// Треугольник и квадрант

**private** **readonly** Quadrant quadrant;

**private** **readonly** Triangle triangle;

// Точки фигуры dego

**private** **readonly** Point d, e, g, o;

// Точки, лежащие на главной диагонали прямоугольника

**private** **readonly** Point a, c;

// Шаг метода

**public** **int** step = 0;

// площадь прямоугольника в котором находится наша фигура dego

**private** **readonly** **double** rectangle\_Square;

// Координаты точки, которую мы будем бросать в прямоугольник

**private** **double** x, y;

//Случайное число от 0 до 1

Random random = **new** Random();

// Точная площадь

**public** **double** dego\_Square;

// Поле в которое будет записана площадь, вычисленная методом Монте-Карло

**public** List<**double**> monte\_carlo\_Square = **new** List<**double**>();

// Относительная погрешность

**public** List<**double**> relError = **new** List<**double**>();

// Поле в которое будет записано время затраченное на метод Calculate

**public** List<**long**> ms = **new** List<**long**>();

// Заполняем наши поля с помощью конструктора

**public** MonteCarloCalculator(Point a, Point e, Point g) // Конструктор инициализации

{

**this**.a = a;

c = **new** Point(e.x + (e.y - a.y), e.y);

d = **new** Point(e.x + (e.y - a.y), a.y);

**this**.e = e;

**this**.g = g;

o = **new** Point(e.x, a.y);

triangle = **new** Triangle(**this**.e, **this**.g, o);

quadrant = **new** Quadrant(o, d);

rectangle\_Square = (c.x - a.x) \* (c.y - a.y);

}

// Метод возвращающий точную площадь dego

**private** **double** GetSquare()

{

**return** triangle.Square()+ quadrant.Square();

}

// Метод возвращающий площадь dego, вычисленную методом Монте-Карло

**private** **double** MonteCarloSquare(**int** N)

{

**int** count = 0;

**for** (**int** i = 0; i < N; i++)

{

x = random.NextDouble() \* (c.x - a.x) + a.x;

y = random.NextDouble() \* (c.y - a.y) + a.y;

Point p = **new** Point(x, y);

**if** (triangle.CheckPoint(p) || quadrant.CheckPoint(p))

{

count++;

}

}

**return** rectangle\_Square \* count / N;

}

**public** **void** Calculate() {

**for** (**int** N = Convert.ToInt32(Math.Pow(10, 3)); N <= Math.Pow(10, 7); N \*= 10) {

Stopwatch sw = **new** Stopwatch();

sw.Start();

// Вычисление точной площади

dego\_Square = GetSquare();

// Вычисление площади методом Монте-Карло

monte\_carlo\_Square.Add(MonteCarloSquare(N));

// Вычисление погрешности вычислений (relative error)

relError.Add(Math.Abs((dego\_Square - monte\_carlo\_Square[step])/dego\_Square)\*100);

// Стоп таймера

sw.Stop();

// Подсчёт милисекунд

ms.Add(sw.ElapsedMilliseconds);

step++;

}

}

}

}

## Файл Table.cs

**using** System;

**using** System.Collections.Generic;

**namespace** ObjectOrientedProgram {

**class** Table {

**int**[] tableCellSizes;

**double** dego\_Square;

List<**double**> monte\_carlo\_Square = **new** List<**double**>();

List<**double**> relError = **new** List<**double**>();

List<**long**> ms = **new** List<**long**>();

**public** Table(MonteCarloCalculator calc) {

tableCellSizes = **new** **int**[] { 14, 24, 30, 26, 12 };

dego\_Square = calc.dego\_Square;

monte\_carlo\_Square = calc.monte\_carlo\_Square;

relError = calc.relError;

ms = calc.ms;

}

**public** **void** DrawTable() {

**for** (**int** i = 0; i < 5; i++) {

DrawHorizontalLine("╔", "╦", "╗", tableCellSizes);

Console.WriteLine("║ N = {0,-8} ║ dego Square = {1,-8} ║ MonteCarlo Square = {2,-8} ║ Relative Error = {3,-6}% ║ ms = {4,-5} ║", Math.Pow(10, 3 + Convert.ToDouble(i)), Math.Round(dego\_Square, 3), Math.Round(monte\_carlo\_Square[i], 3), Math.Round(relError[i], 3), ms[i]);

DrawHorizontalLine("╚", "╩", "╝", tableCellSizes);

}

}

**private** **void** DrawHorizontalLine(**string** a, **string** b, **string** c, **int**[] tableCellSizes) {

Console.Write(a);

**for** (**int** j = 0; j < tableCellSizes.Length; j++) {

**for** (**int** i = 0; i < tableCellSizes[j]; i++) {

Console.Write("═");

}

**if** (j != tableCellSizes.Length - 1) {

Console.Write(b);

}

}

Console.WriteLine(c);

}

}

}

## Файл Triangle.cs

**using** System;

**namespace** ObjectOrientedProgram

{

**class** Triangle

{

// Вершины треугольника

**public** Point p1, p2, p3;

// Стороны треугольника

**public** Side a, b, c;

// Конструктор по умолчанию

**public** Triangle()

{

p1 = **new** Point();

p2 = **new** Point();

p3 = **new** Point();

InitializeSides(p1, p2, p3);

}

// Конструктор инициализации(заполнения)

**public** Triangle(Point p1, Point p2, Point p3)

{

**this**.p1 = p1;

**this**.p2 = p2;

**this**.p3 = p3;

InitializeSides(**this**.p1, **this**.p2, **this**.p3);

}

// Конструктор копирования

**public** Triangle(Triangle t)

{

p1 = t.p1;

p2 = t.p2;

p3 = t.p3;

InitializeSides(t.p1, t.p2, t.p3);

}

// Метод, использующийся для инициализации сторон треугольника

**private** **void** InitializeSides(Point p1, Point p2, Point p3)

{

a = **new** Side(p1, p2);

b = **new** Side(p2, p3);

c = **new** Side(p3, p1);

}

// Метод возвращающий периметр

**public** **double** Perimeter()

{

**return** a.Length() + b.Length() + c.Length();

}

// Метод возвращающий точную площадь треугольника по формуле Герона

**public** **double** Square()

{

**double** p = Perimeter() / 2;

**return** Math.Sqrt(p \* (p - a.Length()) \* (p - b.Length()) \* (p - c.Length()));

}

// Метод проверяющий наличие точки 'p' в треугольнике

**public** **bool** CheckPoint(Point p)

{

**double** buf1 = (p1.x - p.x) \* (p2.y - p1.y) - (p2.x - p1.x) \* (p1.y - p.y);

**double** buf2 = (p2.x - p.x) \* (p3.y - p2.y) - (p3.x - p2.x) \* (p2.y - p.y);

**double** buf3 = (p3.x - p.x) \* (p1.y - p3.y) - (p1.x - p3.x) \* (p3.y - p.y);

**if** ((buf1 >= 0 && buf2 >= 0 && buf3 >= 0) || (buf1 <= 0 && buf2 <= 0 && buf3 <= 0))

{

**return** **true**;

}

**else** **return** **false**;

}

}

}

## Файл Quadrant.cs

**using** System;

**namespace** ObjectOrientedProgram

{

**class** Quadrant

{

// Поля класса

**public** Point center, p;

// Конструктор по умолчанию

**public** Quadrant()

{

center = **new** Point();

p = **new** Point();

}

// Конструктор инициализации(заполнения)

**public** Quadrant(Point center, Point p)

{

**this**.center = center;

**this**.p = p;

}

// Конструктор копирования

**public** Quadrant(Quadrant q)

{

center = q.center;

p = q.p;

}

// Метод возвращающий длину радиуса квадранта

**public** **double** Radius()

{

**return** **new** Side(center, p).Length();

}

// Метод вычисляющий точную площадь квадранта

**public** **double** Square()

{

**return** 0.25 \* Math.PI \* Math.Pow(Radius(), 2);

}

// Метод проверящий попадает ли точка 'p' в квадрант

**public** **bool** CheckPoint(Point p)

{

**if** ((Math.Pow(p.x - center.x, 2) + Math.Pow(p.y - center.y, 2) <= Math.Pow(Radius(), 2)) && (p.x >= center.x) && (p.y >= center.y))

{

**return** **true**;

}

**else** **return** **false**;

}

}

}

## Файл Side.cs

**using** System;

**namespace** ObjectOrientedProgram

{

**public** **class** Side

{

**public** Point p1, p2;

**public** Side() // Конструктор по умолчанию

{

p1 = **new** Point();

p2 = **new** Point();

}

**public** Side(Point p1, Point p2) // Конструктор инициализации(заполнения)

{

**this**.p1 = p1;

**this**.p2 = p2;

}

**public** Side(Side s) // Конструктор копирования

{

p1 = s.p1;

p2 = s.p2;

}

**public** **double** Length() // Метод вычисляющий длину стороны

{

**return** Math.Sqrt(Math.Pow(p1.x - p2.x, 2) + Math.Pow(p1.y - p2.y, 2));

}

}

}

## Файл Point.cs

**namespace** ObjectOrientedProgram

{

**public** **class** Point

{

**public** **double** x, y; // Поля класса

**public** Point() // Конструктор по умолчанию

{

x = 0;

y = 0;

}

**public** Point(**double** x, **double** y) // Конструктор инициализации(заполнения)

{

**this**.x = x;

**this**.y = y;

}

**public** Point(Point p) // Конструктор копирования

{

x = p.x;

y = p.y;

}

}

}