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](#_Toc53138516)

[1. ПОСТАНОВКА ЗАДАЧИ 6](#_Toc53138517)

[2. РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ 9](#_Toc53138518)

[2.1. Анализ вариантов использования программы 9](#_Toc53138519)

[2.2. Требования к приложению 10](#_Toc53138520)

[3. АНАЛИЗ АЛГОРИТМА РЕШЕНИЯ 12](#_Toc53138521)

[3.1. Метод Монте-Карло 12](#_Toc53138522)

[3.2. Определение попадания точки внутрь фигуры 13](#_Toc53138523)

[4. ПРОЦЕДУРНОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C# 14](#_Toc53138524)

[4.1. Программные средства разработки 14](#_Toc53138525)

[4.2. Логическое проектирование 14](#_Toc53138526)

[4.3. Системные требования 14](#_Toc53138527)

[4.4. Структура приложения 15](#_Toc53138528)

[4.5. Структура приложения 18](#_Toc53138529)

[4.6. Результаты работы 18](#_Toc53138530)

[5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C# 20](#_Toc53138531)

[5.1. Программные средства разработки 20](#_Toc53138532)

[5.2. Логическое проектирование 20](#_Toc53138533)

[5.3. Описание программы 21](#_Toc53138534)

[5.4. Руководство системного программиста 29](#_Toc53138535)

[5.5. Руководство программиста 31](#_Toc53138536)

[5.6. Руководство оператора 33](#_Toc53138537)

[5.7. Результаты работы 35](#_Toc53138538)

[6. АНАЛИЗ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ ПРИЛОЖЕНИЙ 37](#_Toc53138539)

[7. УЛУЧШЕНИЕ ХАРАКТЕРИСТИК КАЧЕСТВА ПРИЛОЖЕНИЙ 38](#_Toc53138540)

[7.1. Использование встроенной оптимизации кода 38](#_Toc53138541)

[7.2. Профилирование 38](#_Toc53138542)

[7.3. Результаты улучшения характеристик 40](#_Toc53138543)

[ЗАКЛЮЧЕНИЕ 41](#_Toc53138544)

[СПИСОК ЛИТЕРАТУРЫ 42](#_Toc53138545)

[ПРИЛОЖЕНИЕ 43](#_Toc53138546)

[Файл Program.cs (процедурное приложение) 43](#_Toc53138547)

[Файл Program.cs (объектно-ориентированное приложение) 48](#_Toc53138548)

[Файл Menu.cs 48](#_Toc53138549)

[Файл MonteCarloCalculator.cs 51](#_Toc53138550)

[Файл Triangle.cs 52](#_Toc53138551)

[Файл Quadrant.cs 53](#_Toc53138552)

[Файл Side.cs 54](#_Toc53138553)

[Файл Point.cs 54](#_Toc53138554)

# 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).

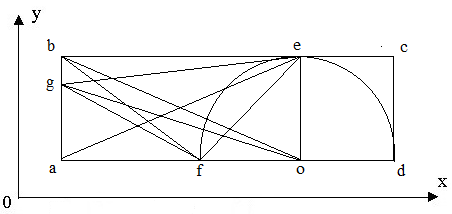


Рис. 1. Фигура *dego*

Исходными данными для каждого приложения являются координаты точек *a,c*, определяющие расположение и размеры прямоугольника в котором расположена наша фигура, расстояние точки *g* от точки *a* и кол-во бросков *N*. Для определения площади фигуры необходимо вначале вычислить площадь прямоугольника, описывающего заданную фигуру, а затем N раз сгенерировать по два случайных числа для координат *x* и *y*, определяющие точку внутри прямоугольника. Генерируемые случайным образом точки 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 Framework;
* 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 выполнения следующих команд: ввод, расчёт, выход.
* возможность получения координат из контрольного примера или из консоли, куда пользователь будет вводить значения.
* координаты представляют собой пару значений *x* и *y* для каждой точки фигуры.
* проверка корректности вводимых данных при вводе значений. Также проверяется, соответствуют ли вершины форме фигуры dego.
* после нахождения площади треугольника в консоль выводится количество точек *N*, количество попавших точек, найденную площадь, погрешность и время работы для *N =* 103, 104, 105, 106, 107.

2.2.2. Требования к процедурному приложению

* 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

* 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) определяются прямоугольником в котором фигуры находятся, а для точки *g* будет определён отступ от точки *a*.

Две точки прямоугольника, лежащие одной из диагоналей, задаются пользователем. Это будет достаточным условием для определения размера и положения прямоугольника.

## 3.1. Метод Монте-Карло

Впишем исходную фигуру в прямоугольник *abec*, площадь которого:

*Sabec = (ex – ax) ∙ (ey – ay)*.

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:

*SMK* = *Sabec.*

Для нахождения точной площади треугольника используется формула Герона:

,

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

*p* = ;

;

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

.

Для нахождения точной площади квадранта используем формулу:

;

Где R определяется точками *o* и *d.*

## 3.2. Определение попадания точки внутрь фигуры

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

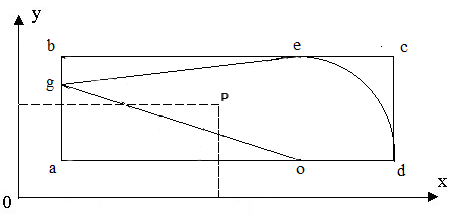


Рис. 3. Точки, необходимые для расчётов

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 рассматриваемую точку  
Если они одинакового знака, то точка внутри треугольника, если что-то из этого - 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, рассматриваемой точки, a,b – координаты точки центра окружности, а R – радиус окружности (od).

В данном случае мы имеем дело с квадрантом из чего следует, что по мимо условиянам потребуется ограничить нашу область двумя прямыми od и oe, то есть требуются условия , .

# ПРОЦЕДУРНОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C#

## 4.1. Программные средства разработки

Процедурное приложение выполняется в инструментальной системе MS Visual Studio 17. Она предоставляет системную объектно-ориентированную среду на базе платформы Microsoft .NET Framework 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 Framework 4.5.2 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:

public void Main() – здесь прописана логика консольного меню, для отрисовки которого, используется метод DrawMenu и один из пунктов меню – Calculate вызывает метод Solution, реализующий расчёт данных и вывод их в консоль в виде таблицы, данные для реализации метода Solution мы вводим используя другие пункты нашего меню (Input Data, Set Example of Data) Если введённые данные введены не корректно или не введены вовсе, то приложение не примет попросит ввести корректные данные, указывая на то, где ошибся пользователь.

public static void Solution() – получает две координаты точек прямоугольника, находящихся на одной из его диагоналей, отступ точки *g* от точки *а* и количество бросков точек *N*, далее метод в цикле с кол-во итераций равным *N* смотрит попала ли случайная точка из пространства прямоугольника в треугольник или квадрант из которых состоит наша фигура dego, используя функции CheckPinT и CheckPinQ, если точка попала , то счётчик увеличивается на 1, а далее используя эти данные считается площадь по методу Монте-Карло, также в этом методе вычисляется площадь фигуры за счёт методов QS и TS, которые используя правила геометрии опять же считают площадь нашей фигуры, и на основе уже этих данных мы можем вычислить погрешность наших вычислений, и в конце метода мы выводим все эти данные в консоль.

Более детально алгоритм можно рассмотреть на рис. 5.

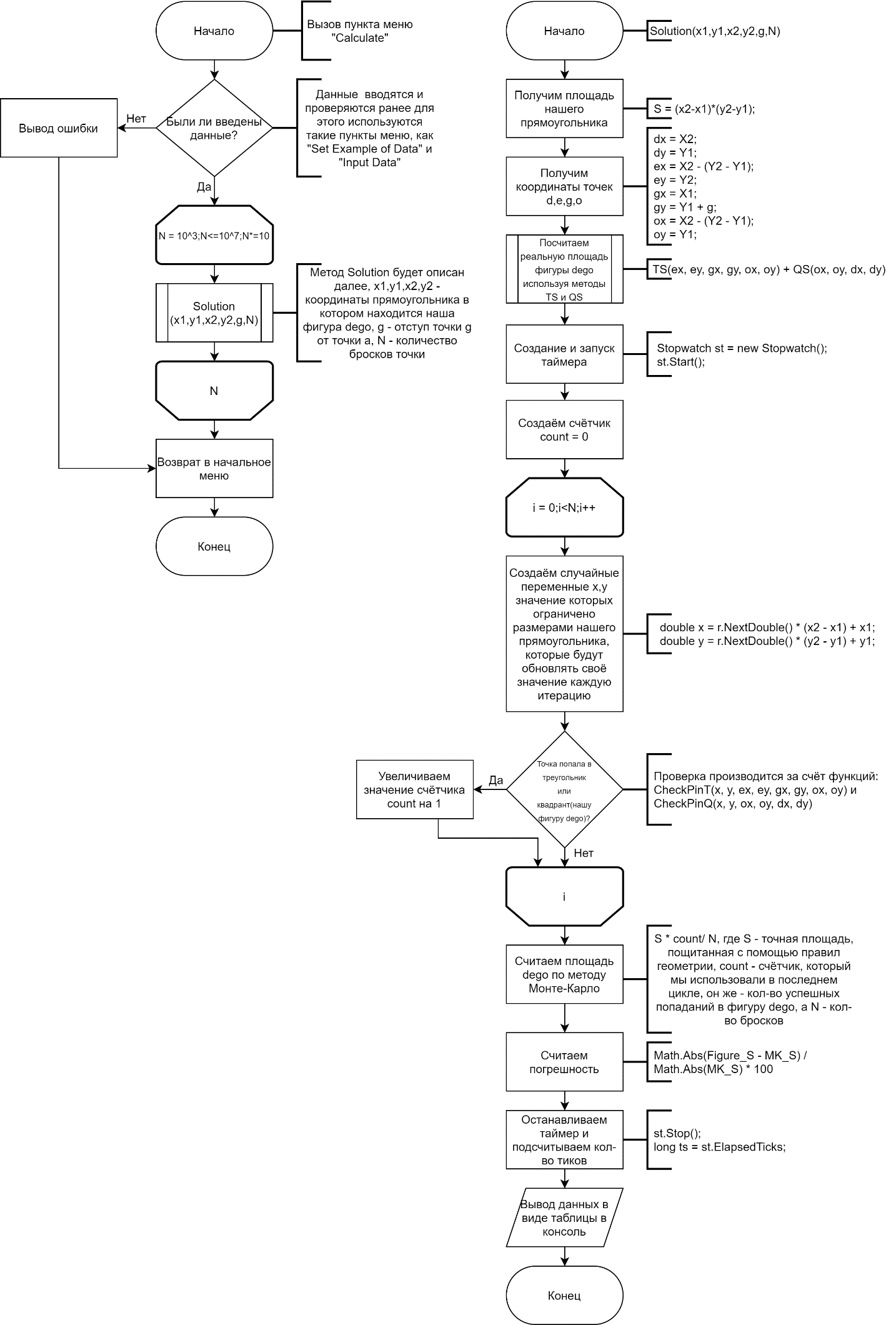


Рис. 5. Схемы метода Solution и его вызова внутри меню

## 4.5. Структура приложения

Структура проекта представлена на рисунке 6.

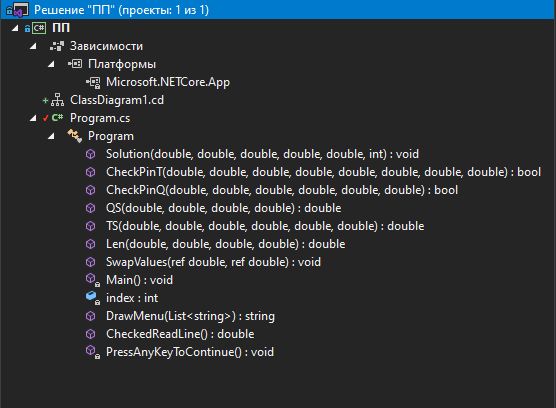


Рис. 6. Проект процедурного приложения

Процедурное приложение включает следующие файлы:

Microsoft.NetCore.App - это набор сборок (DLL-файлы), которые установлены на компьютере, содержащие компоненты среды выполнения и целевой пакет.

ClassDiagram1.cd – файл, содержащий диаграмму классов;

Program.cs – файл, являющийся точкой входа и содержащий все основные методы приложения.

## 4.6. Результаты работы

Результаты работы приложения представлены на рис. 7.

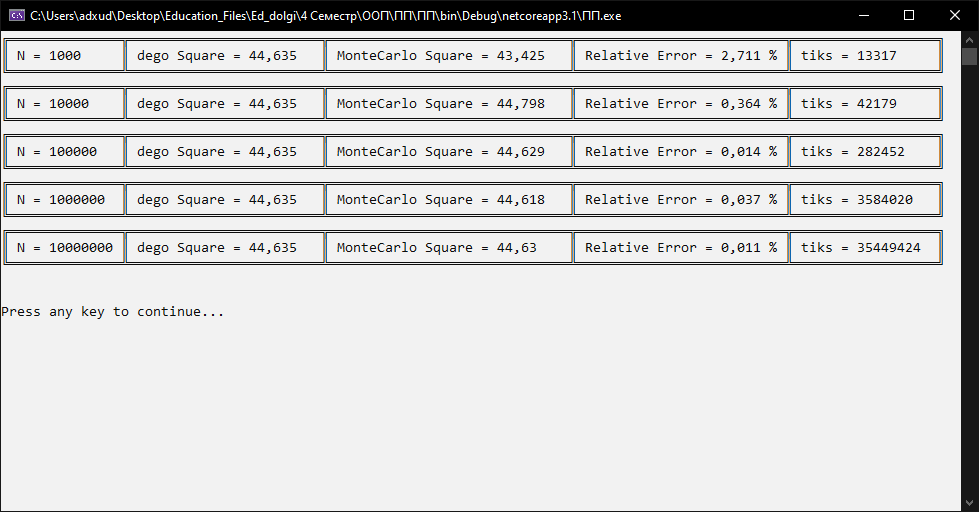


Рис. 7. Тестирование процедурного приложения

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C#

## 5.1. Программные средства разработки

Объектно-ориентированное приложение выполняется в инструментальной системе MS Visual Studio 17. Она предоставляет системную объектно-ориентированную среду на базе платформы Microsoft .NET Framework 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 конструкторы, 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. 8.

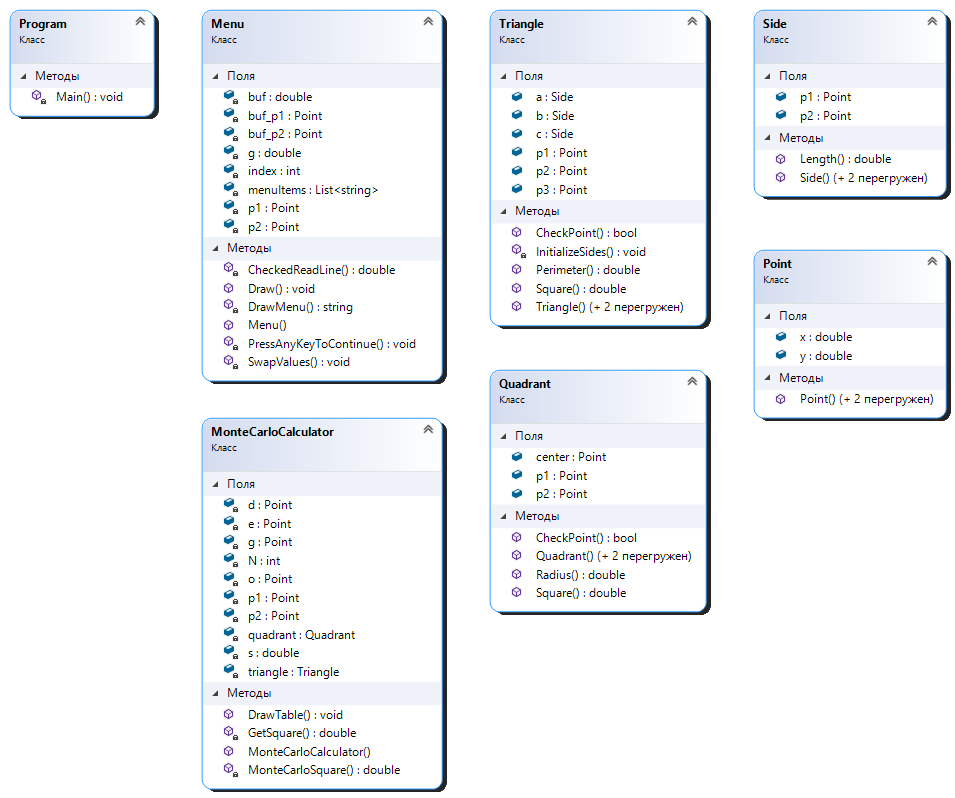


Рис. 8. Диаграмма классов объектно-ориентированного приложения

## 5.3. Описание программы

Класс Point представляет собой точку с двумя координатами. Диаграмма класса представлена на рис. 9.



Рис. 9. 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 две точки и метод для нахождения длины. Диаграмма класса представлена на рис. 10.

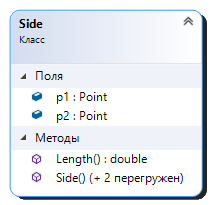


Рис. 10. Диаграмма класса Side

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 три точки с методами по поиску точной площади, периметра, проверки наличия точки внутри треугольника и инициализации сторон, используемый в конструкторах. Диаграмма класса представлена на рис. 11.

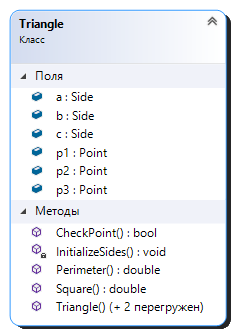


Рис. 11. Диаграмма класса Triangle

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 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 три точки, но для инициализации необходимо лишь две: center и p1. Присутствуют методы по поиску точной площади, радиуса и проверки наличия точки внутри квадранта. Диаграмма класса представлена на рис. 12.

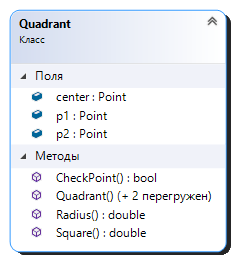


Рис. 12. Диаграмма класса Quadrant

Класс Quadrant содержит следующие поля:

Point center, p1, p2 – центр и 2 точки, с помощью, которых можно описать квадрант.

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 радиус квадранта.

public double Square()– возвращает точную площадь квадранта

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

Класс MonteCarloCalculator представляет собой объект для просчёта площади методом Монте-Карло, реальной площади, относительной погрешности, подсчёта времени и вывода таблицы с результатами расчётов. Диаграмма класса представлена на рис. 13.



Рис. 13. 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 p1, p2 – точки, лежащие на главной диагонали прямоугольника в котором находится наша фигура.

int N – кол-во бросков точки.

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.

Класс Quadrant содержит следующие конструкторы:

public MonteCarloCalculator(Point p1, Point p2, double g, int N)– конструктор инициализации, где входными данными будут точки p1 и p2, отступ точки g от точки a и количество бросков точки для метода Монте-Карло (все данные были заранее проверены и отвергнуты/откорректированы 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 DrawTable() – метод отвечает вынос полученных данных в виде таблицы в консоль.

Класс Menu представляет собой объект, обеспечивающий графический интерфейс для ввода, проверки и автоматической корректировки данных, вводимых пользователем, ввода заранее заготовленного примера данных, создания экземпляра класса MonteCarloCalculator, который и будет осуществлять все расчёты и также для выхода из программы. Диаграмма класса представлена на рис. 14.

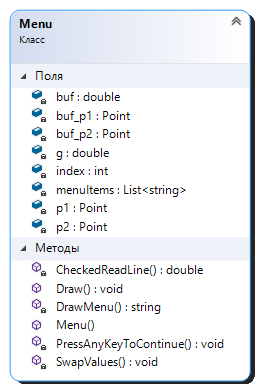


Рис. 14. Диаграмма класса Menu

Класс Menu содержит следующие поля:

Point p1, p2 – точки, лежащие на главной диагонали, прямоугольника в который заключена наша фигура.

double g – расстояние отступ точки g от точки а.

Point buf\_p1, buf\_p2 – буферные поля, точки, необходимые для проверки и автоматической корректировки данных.

double buf – буферное поле, необходимое для проверки и автоматической корректировки данных.

List<string> menuItems – список пунктов меню.

int index – определяет на каком пункте меню пользователь в данный момент находится.

Класс Menu содержит следующие конструкторы:

public Menu() – стандартный конструктор, где прописаны пункты меню.

Класс Menu содержит следующие методы:

private void PressAnyKeyToContinue() – вызывает сообщение “Press any key to continue…” и ставит программу на ожидание нажатия клавиши.

private static double CheckedReadLine() – метод отвечает за ввод и проверку строки на то является ли она числом.

private static string DrawMenu(List<string> items) – метод отвечает за то, как будет выглядеть и как работать меню.

private static void SwapValues(ref double a, ref double b) – метод меняет местами значения переменных.

public void Draw() – метод запускает цикл, обеспечивающий работу меню, а также описывает правила по которым будут работать пункты меню.

Класс Program за счёт метода Main создаёт экземпляр класса меню и вызывает метод этого класса Draw. Диаграмма класса представлена на рис. 15.

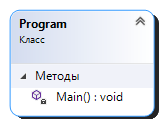


Рис. 15. Диаграмма класса Program

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 Framework 4.5.2 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 работу меню. С помощью пунктов меню «Set Input Data» и «Set Example of Data» выполняется заполнение полей, необходимых для создания экземпляров класса MonteCarloCalculator и вызова метода DrawTable, обеспечивающего построение таблицы с необходимыми данными. Создание и вызов происходит за счёт пункта меню «Calculate». 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 координаты фигуры по умолчанию. В результатах вычисления программы площадь фигуры должна составлять 44,625 единиц.

**5.4.5. Сообщения системному программисту**

В ходе выполнения программы возможен вывод следующих сообщений об ошибках:

* Error: You got a line, not a Rectangle. Please, enter correct data;
* Error: You got a point, not a Rectangle. Please, enter correct data;
* Error: You got a quadrate, not a Rectangle. Please, enter correct data.

При появлении данных ошибок программа предложит повторно ввести координаты точек x и y прямоугольника, в котором находится наша фигура. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* Error: indent more then side AB or less then/equal 0

Please, set indent for point 'g' relative to point 'a' again:

При появлении этой ошибки программа попросит пользователя повторно ввести отступ точки g от точки a, потому что отступ либо больше стороны в которую заключён либо меньше/равен нулю. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* You don't set data / Your data is incorrect

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» для самостоятельного ввода данных или пунктом «Set Example of Data» для использования данных из примера.

* Incorrect data format, please enter data again:

Данная ошибка возникает, если в поле для ввода информации был введён неверный формат данных, далее программа предложит ввести данные ещё раз.

В ходе выполнения программы возможен вывод следующих сообщений:

* I'm swap X and Y axis

Данное сообщение возникает, если при вводе данных пользователь не правильно расположил прямоугольник на координатной плоскости и сообщает о том, что поменяла оси X и Y местами.

* Press any key to continue...

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

## 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;
* «Set Example of Data» - использование данных из 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 Framework 4.5.2 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 1.5 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: You got a line, not a Rectangle. Please, enter correct data;
* Error: You got a point, not a Rectangle. Please, enter correct data;
* Error: You got a quadrate, not a Rectangle. Please, enter correct data.

При появлении данных ошибок программа предложит повторно ввести координаты точек x и y прямоугольника, в котором находится наша фигура. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* Error: indent more then side AB or less then/equal 0

Please, set indent for point 'g' relative to point 'a' again:

При появлении этой ошибки программа попросит пользователя повторно ввести отступ точки g от точки a, потому что отступ либо больше стороны в которую заключён либо меньше/равен нулю. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* You don't set data / Your data is incorrect

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» для самостоятельного ввода данных или пунктом «Set Example of Data» для использования данных из примера.

* Incorrect data format, please enter data again:

Данная ошибка возникает, если в поле для ввода информации был введён неверный формат данных, далее программа предложит ввести данные ещё раз.

В ходе выполнения программы возможен вывод следующих сообщений:

* I'm swap X and Y axis

Данное сообщение возникает, если при вводе данных пользователь не правильно расположил прямоугольник на координатной плоскости и сообщает о том, что поменяла оси X и Y местами.

* Press any key to continue...

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

## 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;
* «Set Example of Data» - использование данных из 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 Framework 4.5.2 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;
* «Set Example of Data» - использование данных из 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: You got a line, not a Rectangle. Please, enter correct data;
* Error: You got a point, not a Rectangle. Please, enter correct data;
* Error: You got a quadrate, not a Rectangle. Please, enter correct data.

При появлении данных ошибок программа предложит повторно ввести координаты точек x и y прямоугольника, в котором находится наша фигура. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* Error: indent more then side AB or less then/equal 0

Please, set indent for point 'g' relative to point 'a' again:

При появлении этой ошибки программа попросит пользователя повторно ввести отступ точки g от точки a, потому что отступ либо больше стороны в которую заключён либо меньше/равен нулю. Ошибка будет повторяться пока пользователь не введёт корректные данные.

* You don't set data / Your data is incorrect

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» для самостоятельного ввода данных или пунктом «Set Example of Data» для использования данных из примера.

* Incorrect data format, please enter data again:

Данная ошибка возникает, если в поле для ввода информации был введён неверный формат данных, далее программа предложит ввести данные ещё раз.

В ходе выполнения программы возможен вывод следующих сообщений:

* I'm swap X and Y axis

Данное сообщение возникает, если при вводе данных пользователь не правильно расположил прямоугольник на координатной плоскости и сообщает о том, что поменяла оси X и Y местами.

* Press any key to continue...

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

## 5.7. Результаты работы

Результаты работы приложения представлены на рис. 16.

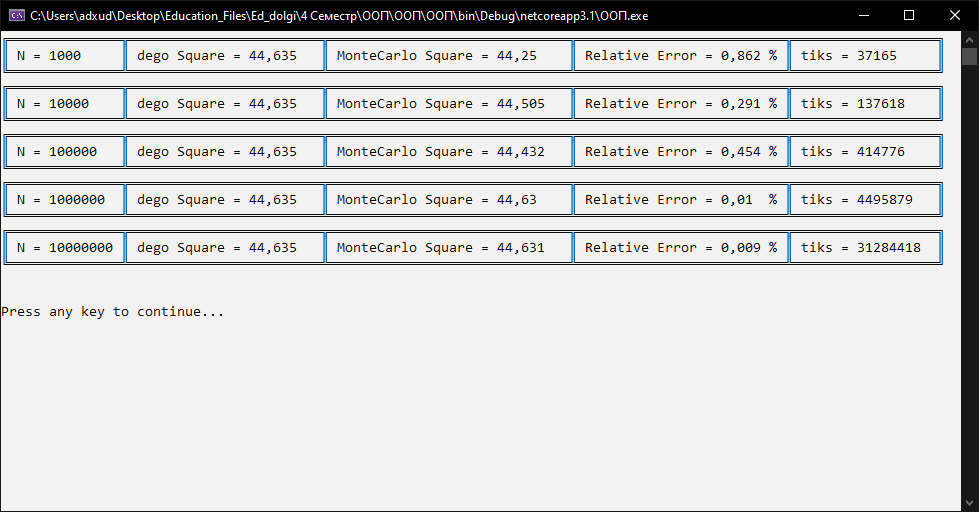


Рис. 16. Тестирование объектно-ориентированного приложения

# 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, тики | |
| Процедурное приложение на базе Console/C# | Объектно-ориентированное приложение на базе Console/C# |
|  | 13317 | 37165 |
|  | 42179 | 137618 |
|  | 282452 | 414776 |
|  | 3584020 | 4495879 |
|  | 35449424 | 31284418 |

В среднем оба предложения обладают одинаковой скоростью работы.

# 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. 17), что в итоге незначительно увеличит производительность.

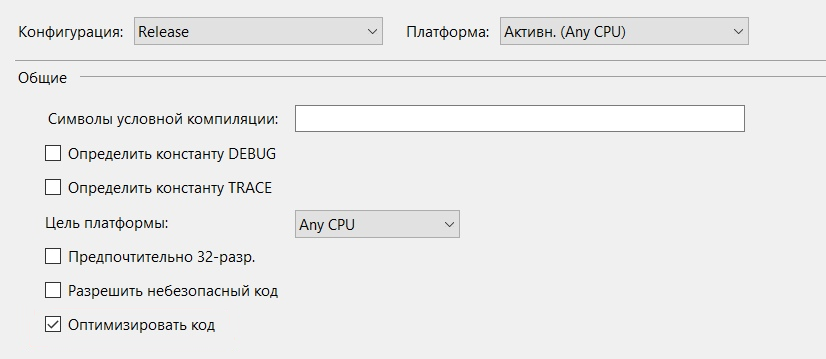


Рис. 17. Вкладка “Сборка” в свойствах проекта

## 7.2. Профилирование

Для приложений было проведено профилирование по использованию ЦП. Результаты приведены на рис. 18 и 19.

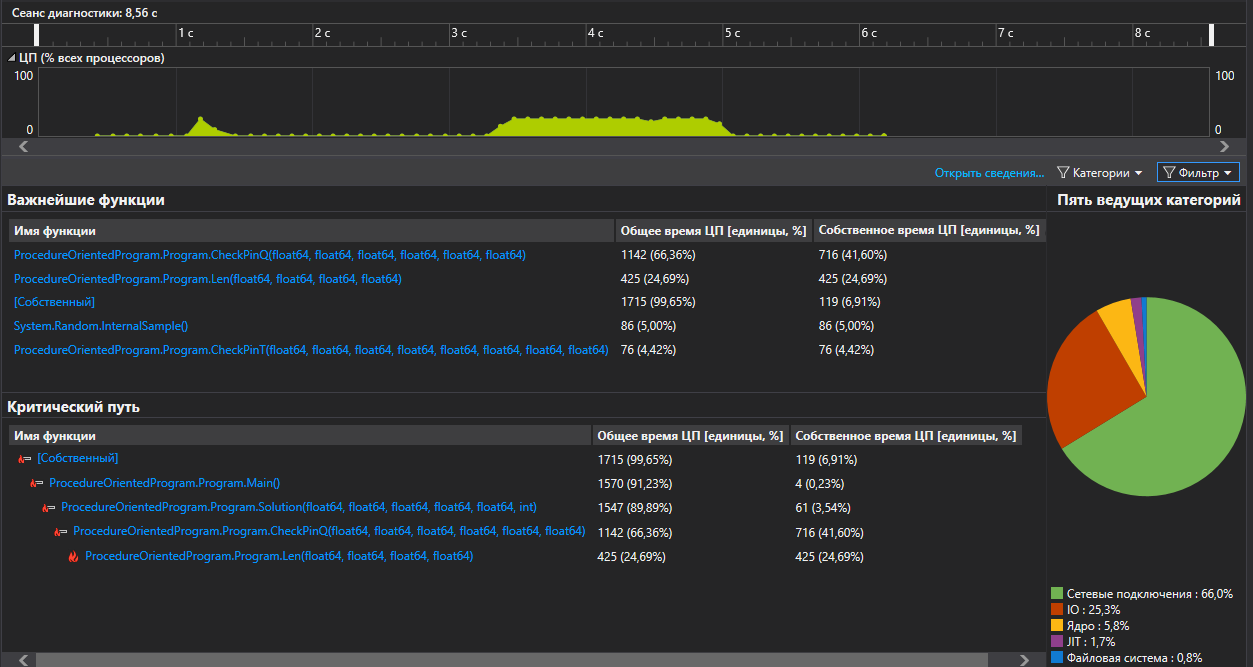


Рис. 18. Результаты профилирования процедурного приложения

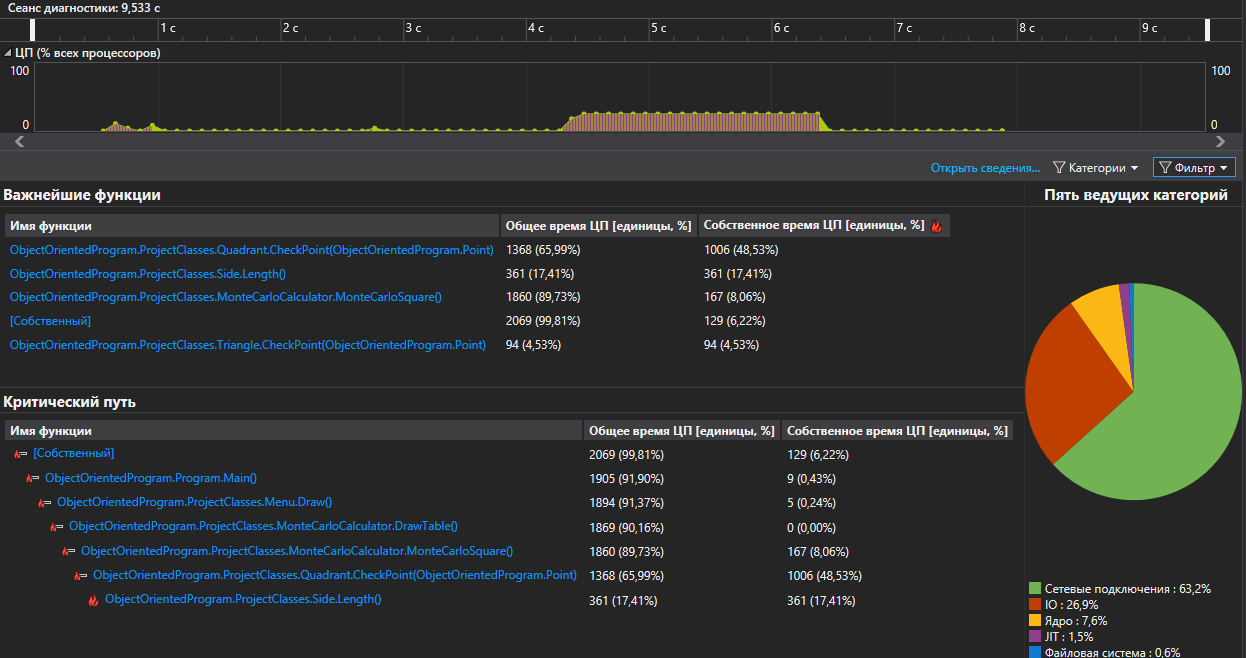


Рис. 19. Результаты профилирования объектно-ориентированного приложения

Из результатов видно, что больше всего ресурсов уходит на метод Menu.Draw(), MonteCarloCalculator.MonteCarloSquare() и MonteCarloCalculator.DrawTable() в объектно-ориентированном приложении, а в процедурном метод Solution(). Также оба приложения примерно одинаково нагружают систему.

## 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, тики | | | |
| Процедурное приложение на базе Console/C# | | Объектно-ориентированное приложение на базе Console/C# | |
| До оптим. | После оптим. | До оптим. | После оптим. |
|  | 13317 | 30195 | 37165 | 1891 |
|  | 42179 | 33720 | 137618 | 16562 |
|  | 282452 | 155019 | 414776 | 185884 |
|  | 3584020 | 1714921 | 4495879 | 1793692 |
|  | 35449424 | 16823986 | 31284418 | 18167048 |

В среднем процедурное приложение стало работать на 52% быстрее, а объектно-ориентированное приложение на 44% 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 Framework; 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 27.05.2020).
2. Metanit [Электронный ресурс] : Полное руководство по языку программирования С# 8.0 и платформе .NET Core 3. Режим доступа: https://metanit.com/sharp/tutorial/ (дата обращения 27.05.2020).
3. Wikipedia [Электронный ресурс] : Метод Монте-Карло. Режим доступа: https://ru.wikipedia.org/wiki/Метод\_Монте-Карло (дата обращения 27.05.2020).
4. Wikipedia [Электронный ресурс] : UML. Режим доступа: https://ru.wikipedia.org/wiki/ UML (дата обращения 27.05.2020).
5. Троелсен Э., Джепикс Ф. Язык программирования C# 7 и платформы. NET и. NET Core. – Litres, 2019.

# ПРИЛОЖЕНИЕ

## Файл Program.cs (процедурно ориентированное приложение)

**using** System;

**using** System.Collections.Generic;

**using** System.Diagnostics;

**namespace** ПП

{

**class** Program

{

// Начало кода основного метода, осуществляющего все расчёты и выведение таблицы в консоль, конечно, используя в том числе и внешние методы

**public** **static** **void** Solution(**double** x1, **double** y1, **double** x2, **double** y2, **double** g, **int** N)

{

**double** S = (x2 - x1) \* (y2 - y1); //Площадь прямоугольника в котором находятся фигуры

// Задаём координаты точек dego

**double** dx = x2;

**double** dy = y1;

**double** ex = x2 - (y2 - y1);

**double** ey = y2;

**double** gx = x1;

**double** gy = y1 + g;

**double** ox = x2 - (y2 - y1);

**double** oy = y1;

**double** figure\_S = TS(ex, ey, gx, gy, ox, oy) + QS(ox, oy, dx, dy); // Площадь фигуры dego, вычисленная с помощью правил геометрии

Stopwatch st = **new** Stopwatch();// объявляем таймер

st.Start();// старт таймера

Random r = **new** Random();

**int** count = 0;

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

{

**double** x = r.NextDouble() \* (x2 - x1) + x1; // Случайная координата x ограниченная размерами прямоугольника

**double** y = r.NextDouble() \* (y2 - y1) + y1; // Случайная координата y ограниченная размерами прямоугольника

**if** (CheckPinT(x, y, ex, ey, gx, gy, ox, oy) || CheckPinQ(x, y, ox, oy, dx, dy)) //Если точка попала в треугольник или квадрант(нашу фигуру dego), то счётчик увеличивается на 1

{

count++;

}

}

**double** mc\_S = S \* count / N;

/\* MK\_S - Площадь фигуры, вычисленная с помощью метода Монте-Карло

            Площадь прямоугольника(ограниченного пространства куда мы будем кидать точки) умножить на кол-во попаданий и поделить на кол-во бросков\*/

**double** relError = Math.Abs((figure\_S - mc\_S) / figure\_S)\*100; // Вычисление погрешности вычислений (relative error)

st.Stop();// стоп таймера

**long** tc = st.ElapsedTicks;// подсчёт тиков

// Вывод таблицы в консоль

**int**[] tableCellSizes = **new** **int**[] { 14, 24, 30, 26, 18 };

**static** **void** DrawHorizontal(**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);

}

**void** DrawCellLine(**int**[] tableCellSizes)

{

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

Console.WriteLine("║ N = {0,-8} ║ dego Square = {1,-8} ║ MonteCarlo Square = {2,-8} ║ Relative Error = {3,-6}% ║ tiks = {4,-9} ║", N, Math.Round(figure\_S, 3), Math.Round(mc\_S, 3), Math.Round(relError, 3), tc);

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

}

DrawCellLine(tableCellSizes);

}

//Далее идут различные методы, использованные в коде выше

//Проверка наличия точки в Треугольнике

**public** **static** **bool** CheckPinT(**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

**public** **static** **bool** CheckPinQ(**double** x, **double** y, **double** cntrX, **double** cntrY, **double** Px, **double** Py)

{

**double** R = Len(cntrX, cntrY, Px, Py);

**if** ((Math.Pow(x - cntrX, 2) + Math.Pow(y - cntrY, 2) <= Math.Pow(R, 2)) && (x >= cntrX) && (y >= cntrY))

{

**return** **true**;

}

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

}

// Вычисление площади квадранта с помощью правил геометрии

**public** **static** **double** QS(**double** X1, **double** Y1, **double** X2, **double** Y2)

{

**return** Math.Pow(Len(X1, Y1, X2, Y2), 2) \* 0.25 \* Math.PI;

}

// Вычисление площади треугольника с помощью правил геометрии

**public** **static** **double** TS(**double** X1, **double** Y1, **double** X2, **double** Y2, **double** X3, **double** Y3)

{

**double** P = (Len(X1, Y1, X2, Y2) + Len(X2, Y2, X3, Y3) + Len(X3, Y3, X1, Y1)) / 2;

**return** Math.Sqrt(P \* (P - Len(X1, Y1, X2, Y2)) \* (P - Len(X2, Y2, X3, Y3)) \* (P - Len(X3, Y3, X1, Y1)));

}

// Вычисление длины отрезка

**public** **static** **double** Len(**double** X1, **double** Y1, **double** X2, **double** Y2)

{

**return** Math.Sqrt(Math.Pow(X1 - X2, 2) + Math.Pow(Y1 - Y2, 2));

}

// Метод меняющий значения переменных местами

**public** **static** **void** SwapValues(**ref** **double** a, **ref** **double** b)

{

a += b;

b = a - b;

a -= b;

}

// Отсюда начинается Меню

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

{

// Введение переменных, необходимых для описания фигуры и буферных переменных, необходимых для предотвращения ошибок при вводе данных

**double** x1 = 0;

**double** x2 = 0;

**double** y1 = 0;

**double** y2 = 0;

**double** g = 0;

**double** bx1, by1, bx2, by2, buf;

Console.CursorVisible = **false**;

List<**string**> menuItem = **new** List<**string**>() // Создание пунктов меню

{

"Set Input Data",

"Set Example of Data",

"Show Current Data",

"Calculate",

"Exit",

""

};

//Цикл вызывающий определённые действия в зависимости от выбранного пункта меню

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

{

**string** SelectedMenuItem = DrawMenu(menuItem);

// 1.

// Пункт меню, запускающий ввод данных вручную

**if** (SelectedMenuItem == "Set Input Data")

{

Console.Clear();

SetRectangle();

**void** SetRectangle() //Метод для введения введения координат прямоугольника, содержащего нашу фигуру, с помощью 2-ух точек, лежащих на его диагонали

{

Console.WriteLine("Set coordinates for 1st point on diagonal of your Rectangle\n\n");

Console.Write("Set X:");

bx1 = CheckedReadLine();

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

by1 = CheckedReadLine();

Console.Clear();

Console.WriteLine("Set coordinates for 2nd point on diagonal of your Rectangle\n\n");

Console.Write("Set X:");

bx2 = CheckedReadLine();

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

by2 = CheckedReadLine();

Console.Clear();

}

**while** (bx1 == bx2 || by1 == by2) // Вывод ошибки о том, что по таким точкам можно построить только линию и просьба ввести данные заного

{

Console.WriteLine("Error: You got a line, not a Rectangle\n\nPlease, enter correct data");

PressAnyKeyToContinue();

SetRectangle();

}

**while** (bx1 == bx2 && by1 == by2) // Вывод ошибки о том, что по таким точкам можно построить только точку

{

Console.WriteLine("Error: You got a point, not a Rectangle\n\nPlease, enter correct data");

PressAnyKeyToContinue();

SetRectangle();

}

**while** (Math.Abs(bx1 - bx2) == Math.Abs(by1 - by2)) // Вывод ошибки о том, что по таким точкам можно построить только квадрат

{

Console.WriteLine("Error: You got a quadrate, not a Rectangle\n\nPlease, enter correct data");

PressAnyKeyToContinue();

SetRectangle();

}

//Если пользователь не правильно расположил прямоугольник программа сама исправит это поменяв координаты X и Y

**if** ((bx1 > bx2) && (by1 > by2)) { SwapValues(**ref** bx1, **ref** bx2); SwapValues(**ref** by1, **ref** by2); }

**else** **if** ((bx1 < bx2) && (by1 > by2)) { SwapValues(**ref** by1, **ref** by2); }

**else** **if** ((bx1 > bx2) && (by1 < by2)) { SwapValues(**ref** bx1, **ref** bx2); }

**if** (Math.Abs(bx1 - bx2) <= Math.Abs(by1 - by2))

{

SwapValues(**ref** bx1, **ref** by1);

SwapValues(**ref** bx2, **ref** by2);

Console.WriteLine("I'm swap X and Y axis");

PressAnyKeyToContinue();

}

// Просьба ввести расстояние от точки g до точки a

Console.Write("Set indent for point 'g' relative to point 'a': ");

buf = CheckedReadLine();

// Проверка того, что расстояние от точки g до точки не больше, чем сторона ab

**while** (buf >= Math.Abs(by2 - by1) || buf <= 0)

{

Console.Clear();

Console.WriteLine("Error: indent more then side AB or less then/equal 0");

Console.Write("Please, set indent for point 'g' relative to point 'a' again: ");

buf = CheckedReadLine();

}

x1 = bx1;

x2 = bx2;

y1 = by1;

y2 = by2;

g = buf;

Console.Clear();

Console.WriteLine("Your Rectangle was set\nP1: X={0}, Y={1};\nP2: X={2}, Y={3}\nIndent for point 'g': {4}.", x1, y1, x2, y2, g);

PressAnyKeyToContinue();

}

// 2.

// Пункт меню с помощью которого мы можем задать пример данных для построения фигуры

**else** **if** (SelectedMenuItem == "Set Example of Data")

{

x1 = 0;

y1 = 0;

x2 = 15;

y2 = 5;

g = 3;

Console.Clear();

Console.Write("You set Example of Data\nP1: X={0}, Y={1};\nP2: X={2}, Y={3}\nIndent for point 'g': {4}.", x1, y1, x2, y2, g);

PressAnyKeyToContinue();

}

// 3.

// Пункт меню, показывающий текущие данные для построения фигуры

**else** **if** (SelectedMenuItem == "Show Current Data")

{

Console.Clear();

Console.WriteLine("Your Rectangale current data\nP1: X={0}, Y={1};\nP2: X={2}, Y={3}\nIndent for point 'g': {4}.", x1, y1, x2, y2, g);

PressAnyKeyToContinue();

}

// 4.

// Пункт меню запускающий вычисления

**else** **if** (SelectedMenuItem == "Calculate")

{

Console.Clear();

// Проверка на то были ли введены данные

**if** ((x1 == x2) || (y1 == y2))

{

Console.WriteLine("You don't set data / Your data is incorrect");

PressAnyKeyToContinue();

}

**else**

{

**for** (**int** N = 1000; N <= Math.Pow(10, 7); N \*= 10)

{

Solution(x1, y1, x2, y2, g, N); // Вызов нашего основного метода Solution, используя ранее введённые данные

}

PressAnyKeyToContinue();

}

}

// 5.

// Пункт меню, обеспечивающий выход из программы

**else** **if** (SelectedMenuItem == "Exit")

{

Environment.Exit(0);

}

}

}

// Метод DrawMenu. С помощью него и происходит отрисовка меню.

**private** **static** **int** index = 0;

**public** **static** **string** DrawMenu(List<**string**> items)

{

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

{

**if** (i == index)

{

Console.BackgroundColor = ConsoleColor.Gray;

Console.ForegroundColor = ConsoleColor.Black;

Console.WriteLine(items[i]);

}

**else**

{

Console.WriteLine(items[i]);

}

Console.ResetColor();

}

ConsoleKeyInfo ckey = Console.ReadKey();

**if** (ckey.Key == ConsoleKey.DownArrow && index < items.Count - 2)

{

index++;

}

**else** **if** (ckey.Key == ConsoleKey.UpArrow && index > 0)

{

index--;

}

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

{

**return** items[index];

}

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

{

Environment.Exit(0);

}

Console.Clear();

**return** "";

}

// Метод проверяющий является ли строка числом

**public** **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 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();

}

}

}

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

**using** ООП.ProjectClasses;

**namespace** ООП

{

**class** Program

{

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

{

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

}

}

}

## Файл Menu.cs

**using** System;

**using** System.Collections.Generic;

**namespace** ООП.ProjectClasses

{

**class** Menu

{

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

// Введение полей, необходимых для описания фигуры и буферных переменных, необходимых для предотвращения ошибок при вводе данных

**private** Point p1 = **new** Point();

**private** Point p2 = **new** Point();

**private** Point buf\_p1 = **new** Point();

**private** Point buf\_p2 = **new** Point();

**private** **double** g = 0;

**private** **double** buf;

**public** Menu()

{

List<**string**> menuItems = **new** List<**string**>() // Создание пунктов меню

{

"Set Input Data",

"Set Example of Data",

"Show Current Data",

"Calculate",

"Exit"

};

**this**.menuItems = menuItems;

**this**.menuItems.Add("");

}

// Основной метод класса запускающий цикл, обеспечивающий работу меню

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

{

Console.CursorVisible = **false**;

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

{

**string** SelectedMenuItem = DrawMenu(menuItems);

// 1.

// Пункт меню, запускающий ввод данных вручную

**if** (SelectedMenuItem == "Set Input Data")

{

Console.Clear();

SetRectangle();

**void** SetRectangle() //Метод для введения введения координат прямоугольника, содержащего нашу фигуру, с помощью 2-ух точек, лежащих на его диагонали

{

Console.WriteLine("Set coordinates for 1st point on diagonal of your Rectangle\n\n");

Console.Write("Set X:");

buf\_p1.x = CheckedReadLine();

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

buf\_p1.y = CheckedReadLine();

Console.Clear();

Console.WriteLine("Set coordinates for 2nd point on diagonal of your Rectangle\n\n");

Console.Write("Set X:");

buf\_p2.x = CheckedReadLine();

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

buf\_p2.y = CheckedReadLine();

Console.Clear();

}

**while** (buf\_p1.x == buf\_p2.x || buf\_p1.y == buf\_p2.y) // Вывод ошибки о том, что по таким точкам можно построить только линию и просьба ввести данные заново

{

Console.WriteLine("Error: You got a line, not a Rectangle\n\nPlease, enter correct data");

PressAnyKeyToContinue();

SetRectangle();

}

**while** (buf\_p1.x == buf\_p2.x && buf\_p1.y == buf\_p2.y) // Вывод ошибки о том, что по таким точкам можно построить только точку

{

Console.WriteLine("Error: You got a point, not a Rectangle\n\nPlease, enter correct data");

PressAnyKeyToContinue();

SetRectangle();

}

**while** (Math.Abs(buf\_p1.x - buf\_p2.x) == Math.Abs(buf\_p1.y - buf\_p2.y)) // Вывод ошибки о том, что по таким точкам можно построить только квадрат

{

Console.WriteLine("Error: You got a quadrate, not a Rectangle\n\nPlease, enter correct data");

PressAnyKeyToContinue();

SetRectangle();

}

//Если пользователь не правильно расположил прямоугольник программа сама исправит это поменяв координаты X и Y

**if** ((buf\_p1.x > buf\_p2.x) && (buf\_p1.y > buf\_p2.y)) { SwapValues(**ref** buf\_p1.x, **ref** buf\_p2.x); SwapValues(**ref** buf\_p1.y, **ref** buf\_p2.y); }

**else** **if** ((buf\_p1.x < buf\_p2.x) && (buf\_p1.y > buf\_p2.y)) { SwapValues(**ref** buf\_p1.y, **ref** buf\_p2.y); }

**else** **if** ((buf\_p1.x > buf\_p2.x) && (buf\_p1.y < buf\_p2.y)) { SwapValues(**ref** buf\_p1.x, **ref** buf\_p2.x); }

**if** (Math.Abs(buf\_p1.x - buf\_p2.x) <= Math.Abs(buf\_p1.y - buf\_p2.y))

{

SwapValues(**ref** buf\_p1.x, **ref** buf\_p1.y);

SwapValues(**ref** buf\_p2.x, **ref** buf\_p2.y);

Console.WriteLine("I'm swap X and Y axis");

PressAnyKeyToContinue();

}

// Просьба ввести расстояние от точки g до точки a

Console.Write("Set indent for point 'g' relative to point 'a': ");

buf = CheckedReadLine();

// Проверка того, что расстояние от точки g до точки не больше, чем сторона ab

**while** (buf >= Math.Abs(buf\_p2.y - buf\_p1.y) || buf <= 0)

{

Console.Clear();

Console.WriteLine("Error: indent more then side AB or less then/equal 0");

Console.Write("Please, set indent for point 'g' relative to point 'a' again: ");

buf = CheckedReadLine();

}

p1.x = buf\_p1.x;

p2.x = buf\_p2.x;

p1.y = buf\_p1.y;

p2.y = buf\_p2.y;

g = buf;

Console.Clear();

Console.WriteLine("Your Rectangle was set\nP1: X={0}, Y={1};\nP2: X={2}, Y={3}\nIndent for point 'g': {4}.", p1.x, p1.y, p2.x, p2.y, g);

PressAnyKeyToContinue();

}

// 2.

// Пункт меню с помощью которого мы можем задать пример данных для построения фигуры

**else** **if** (SelectedMenuItem == "Set Example of Data")

{

p1.x = 0;

p1.y = 0;

p2.x = 15;

p2.y = 5;

g = 3;

Console.Clear();

Console.Write("You set Example of Data\nP1: X={0}, Y={1};\nP2: X={2}, Y={3}\nIndent for point 'g': {4}.", p1.x, p1.y, p2.x, p2.y, g);

PressAnyKeyToContinue();

}

// 3.

// Пункт меню, показывающий текущие данные для построения фигуры

**else** **if** (SelectedMenuItem == "Show Current Data")

{

Console.Clear();

Console.WriteLine("Your Rectangale current data\nP1: X={0}, Y={1};\nP2: X={2}, Y={3}\nIndent for point 'g': {4}.", p1.x, p1.y, p2.x, p2.y, g);

PressAnyKeyToContinue();

}

// 4.

// Пункт меню запускающий вычисления

**else** **if** (SelectedMenuItem == "Calculate")

{

Console.Clear();

// Проверка на то были ли введены данные

**if** ((p1.x == p2.x) || (p1.y == p2.y))

{

Console.WriteLine("You don't set data / Your data is incorrect");

PressAnyKeyToContinue();

}

**else**

{

**for** (**int** N = 1000; N <= Math.Pow(10, 7); N \*= 10)

{

MonteCarloCalculator calc = **new** MonteCarloCalculator(p1, p2, g, N);

calc.DrawTable();

}

PressAnyKeyToContinue();

}

}

// 5.

// Пункт меню, обеспечивающий выход из программы

**else** **if** (SelectedMenuItem == "Exit")

{

Environment.Exit(0);

}

}

}

// Метод меняющий значения переменных местами

**private** **static** **void** SwapValues(**ref** **double** a, **ref** **double** b)

{

a += b;

b = a - b;

a -= b;

}

// Метод DrawMenu. С помощью него и происходит отрисовка меню.

**private** **static** **int** index = 0;

**private** **static** **string** DrawMenu(List<**string**> items)

{

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

{

**if** (i == index)

{

Console.BackgroundColor = ConsoleColor.Gray;

Console.ForegroundColor = ConsoleColor.Black;

Console.WriteLine(items[i]);

}

**else**

{

Console.WriteLine(items[i]);

}

Console.ResetColor();

}

ConsoleKeyInfo ckey = Console.ReadKey();

**if** (ckey.Key == ConsoleKey.DownArrow && index < items.Count - 2)

{

index++;

}

**else** **if** (ckey.Key == ConsoleKey.UpArrow && index > 0)

{

index--;

}

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

{

**return** items[index];

}

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

{

Environment.Exit(0);

}

Console.Clear();

**return** "";

}

// Метод проверяющий является ли строка числом

**private** **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 сообщающий, что нужно нажать на клавишу для того, чтобы продолжить

**private** **void** PressAnyKeyToContinue()

{

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

Console.ReadKey();

Console.Clear();

}

}

}

## Файл MonteCarloCalculator.cs

**using** System;

**using** System.Diagnostics;

**namespace** ООП.ProjectClasses

{

**public** **class** MonteCarloCalculator

{

// Поля нашего класса

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

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

**private** **readonly** Point d, e, g, o, p1, p2;

**private** **readonly** **int** N;

**private** **readonly** **double** s; // площадь прямоугольника в котором находится наша фигура dego

// Заполняем наши поля с помощью конструктора

**public** MonteCarloCalculator(Point p1, Point p2, **double** g, **int** N) // g - отступ точки g от точки a

{

**this**.p1 = p1;

**this**.p2 = p2;

d = **new** Point(p2.x, p1.y);

e = **new** Point(p2.x - (p2.y - p1.y), p2.y);

**this**.g = **new** Point(p1.x, p1.y + g);

o = **new** Point(p2.x - (p2.y - p1.y), p1.y);

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

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

**this**.N = N;

s = (p2.x - p1.x) \* (p2.y - p1.y);

}

**private** **double** GetSquare() // Получение площади dego

{

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

}

**private** **double** MonteCarloSquare() // Получение площади dego методом Монте-Карло

{

Random rand = **new** Random();

**int** count = 0;

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

{

**double** x = rand.NextDouble() \* (p2.x - p1.x) + p1.x;

**double** y = rand.NextDouble() \* (p2.y - p1.y) + p1.y;

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

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

{

count++;

}

}

**return** s \* count / N;

}

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

{

// Вычисление данных для таблицы

**double** figure\_S = GetSquare();

**double** mc\_S;

**double** relError;

Stopwatch st = **new** Stopwatch();

st.Start();

mc\_S = MonteCarloSquare();

relError = Math.Abs((figure\_S - mc\_S) / figure\_S) \* 100;

st.Stop();

**long** tc = st.ElapsedTicks;

//Непосредственно отрисовка таблицы

**int**[] tableCellSizes = **new** **int**[] { 14, 24, 30, 26, 18 };

**static** **void** DrawHorizontal(**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);

}

**void** DrawCellLine(**int**[] tableCellSizes)

{

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

Console.WriteLine("║ N = {0,-8} ║ dego Square = {1,-8} ║ MonteCarlo Square = {2,-8} ║ Relative Error = {3,-6}% ║ tiks = {4,-9} ║", N, Math.Round(figure\_S, 3), Math.Round(mc\_S, 3), Math.Round(relError, 3), tc);

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

}

DrawCellLine(tableCellSizes);

}

}

}

## Файл Triangle.cs

**using** System;

**namespace** ООП.ProjectClasses

{

**class** Triangle

{

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

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

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

{

p1 = **new** Point();

p2 = **new** Point();

p3 = **new** Point();

InitializeSides(p1, p2, p3);

}

**public** Triangle(Point p1, Point p2, Point p3) // Конструктор инициализации(заполнения)

{

**this**.p1 = p1;

**this**.p2 = p2;

**this**.p3 = p3;

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

}

**public** Triangle(Triangle t) // Конструктор копирования

{

p1 = t.p1;

p2 = t.p2;

p3 = t.p3;

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

}

**private** **void** InitializeSides(Point p1, Point p2, Point p3) // Метод для заоплнения полей сторон

{

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

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

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

}

**public** **double** Perimeter() // Метод возвращающий периметр

{

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

}

**public** **double** Square() // Метод возвращающий площадь треугольника по полупериметру.

{

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

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

}

**public** **bool** CheckPoint(Point p) // Метод проверяющий наличие точки в треугольнике

{

**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** ООП.ProjectClasses

{

**class** Quadrant

{

**public** Point center, p1, p2; // Поля класса

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

{

center = **new** Point();

p1 = **new** Point();

p2 = **new** Point();

}

**public** Quadrant(Point center, Point p1) // Конструктор инициализации(заполнения)

{

**this**.center = center;

**this**.p1 = p1;

p2 = **new** Point(**this**.center.x, **this**.center.y + **new** Side(center, p1).Length());

}

**public** Quadrant(Quadrant q) // Конструктор копирования

{

center = q.center;

p1 = q.p1;

p2 = q.p2;

}

**public** **double** Radius() // Метод возвращающий длину радиуса квадранта

{

**return** **new** Side(center, p1).Length();

}

**public** **double** Square() // Метод считающий площадь квадранта

{

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

}

**public** **bool** CheckPoint(Point p) // Метод проверящий попадает ли точка в квадрант

{

**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** ООП.ProjectClasses

{

**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** ООП

{

**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;

}

}

}