МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

Курсовая работа по дисциплине

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

на тему: Реализация метода Монте-Карло.

Студент группы 220681 Страхова Д.А. \_\_\_\_\_\_\_\_\_\_\_

(Ф.И.О.) (Подпись, дата)

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

(Ф.И.О., должность) (Подпись, дата)

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

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

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

**З а д а н и е**

На курсовой проект (курсовую работу) по дисциплине (наименование

дисциплины указывать полностью) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

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

студенту(ке) группы \_\_\_\_\_\_\_\_\_\_\_ Ф.И.О. студента \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

Тема проекта (работы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

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

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

Входные данные \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

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

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

Задание получил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ «\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_ г.

(подпись студента)

Срок представления задания «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_г.

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

(подпись) (расшифровка подписи)

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

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

(подпись) (расшифровка подписи)

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

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

При защите курсового проекта (работы) наличие рецензии обязательно

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 5](#_Toc42831783)

[1. ПОСТАНОВКА ЗАДАЧИ 7](#_Toc42831784)

[2. РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ 8](#_Toc42831785)

[2.1. Анализ вариантов использования программы 8](#_Toc42831786)

[2.2. Требования к приложениям 9](#_Toc42831787)

[3. АНАЛИЗ АЛГОРИТМА РЕШЕНИЯ 10](#_Toc42831788)

[3.1. Метод Монте-Карло 10](#_Toc42831789)

[3.2. Определение попадания точки внутрь фигуры 10](#_Toc42831790)

[4. ПРОЦЕДУРНОЕ ПРИЛОЖЕНИЕ НА БАЗЕ C# 11](#_Toc42831791)

[4.1. Программные средства разработки 11](#_Toc42831792)

[4.2. Логическое проектирование 11](#_Toc42831793)

[4.3. Системные требования 11](#_Toc42831794)

[4.4. Структура приложения 11](#_Toc42831795)

[4.5. Результаты работы 11](#_Toc42831796)

[5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРИЛОЖЕНИЕ НА БАЗЕ C# 12](#_Toc42831797)

[5.1. Программные средства разработки 12](#_Toc42831798)

[5.2. Логическое проектирование 12](#_Toc42831799)

[5.3. Описание программы 12](#_Toc42831800)

[5.4. Руководство системного программиста 12](#_Toc42831801)

[5.4.1 Общие сведения о программе 12](#_Toc42831802)

[5.4.2 Структура программы 12](#_Toc42831803)

[5.4.3 Настройка программы 12](#_Toc42831804)

[5.4.4 Проверка программы 12](#_Toc42831805)

[5.5. Руководство программиста 12](#_Toc42831806)

[5.5.1 Назначение и условие применения программы 12](#_Toc42831807)

[5.5.2 Характеристика программы 12](#_Toc42831808)

[5.5.3 Обращение к программе 12](#_Toc42831809)

[5.5.4 Входные и выходные данные 12](#_Toc42831810)

[5.6. Руководство оператора 12](#_Toc42831811)

[5.6.1 Назначение программы 12](#_Toc42831812)

[5.6.2 Условия выполнения программы 12](#_Toc42831813)

[5.7. Результаты работы 13](#_Toc42831814)

[6. АНАЛИЗ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ ПРИЛОЖЕНИЙ 13](#_Toc42831815)

[7. УЛУЧШЕНИЕ ХАРАКТЕРИСТИК КАЧЕСТВА ПРИЛОЖЕНИЙ 14](#_Toc42831816)

[7.1. Использование встроенной оптимизации кода 14](#_Toc42831817)

[7.2. Профилирование 14](#_Toc42831818)

[7.3. Результаты улучшения характеристик 14](#_Toc42831819)

[ЗАКЛЮЧЕНИЕ 15](#_Toc42831820)

[СПИСОК ЛИТЕРАТУРЫ 16](#_Toc42831821)

[ПРИЛОЖЕНИЕ 17](#_Toc42831822)

[Файл Program.cs (процедурное приложение) 17](#_Toc42831823)

[Файл Program.cs (объектно-ориентированное приложение) 17](#_Toc42831824)

[Файл Form1.cs (процедурное приложение) 17](#_Toc42831825)

[Файл Form1.cs (объектно-ориентированное приложение) 17](#_Toc42831826)

[Файл SquareClass.cs 17](#_Toc42831827)

# 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 (ПО) 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 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. Расширяемость означает простоту добавления новых функций, сопровождаемость является показателем простоты поиска и устранения ошибок, а переносимость означает простоту переноса программы в другую операционную систему или ее новую версию. Масштабируемость определяет способность приложения работать при увеличении нагрузки, используя предусмотренные для этого средства.

На результатах ООА формируются модели, на которых основывается OOД, а OOД, в свою очередь, создает фундамент для окончательной реализации системы с использованием методологии OOП.

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»)).

Разработка ПО обычно производится с помощью специальных CASE-средств для автоматизированного проектирования и создания программ (computer-aided software engineering - CASE). Накопленный опыт автоматизированного создания программных систем показал, что одной из основных целей разрабатываемых автоматизированных процессов создания и эксплуатации ПО является стремление уменьшить зависимость проектной организации от конкретных исполнителей. Достигается это в первую очередь высокой дисциплинированностью и хорошей документированностью самого процесса проектирования ПО. Появление языка UML значительно способствовало решению этой задачи.

Существует огромное количество методологий и рекомендаций, направленных на повышение эффективности процесса проектирования программных систем. Однако, в основе всех этих методологий лежит универсальная методика - принцип повторного использования, представленный техникой шаблонного проектирования. Понятие "шаблон (паттерн) проектирования" – это описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

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

# ПОСТАНОВКА ЗАДАЧИ

Основная задача курсовой работы – разработать процедурное и объектно-ориентированное приложения, решающие задачу приближенного вычисления площади фигуры методом Монте-Карло, а так же выполнить исследование вычислительной эффективности созданных приложений и представить результаты в виду таблиц, провести анализ и улучшить заданные характеристики качества приложений.

Вариант задания №40:

* геометрическая фигура: *gef*;
* приложение 1: процедурное приложение на базе Windows Forms/C#;
* приложение 2: объектно-ориентированное приложение на базе Windows Forms/C#.

Геометрическая фигура представляет собой треугольник, вписанный в прямоугольник (рис.1)

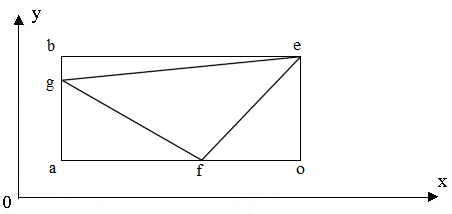


Рис.1. Фигура *gef*

Исходными данными для каждого приложения являются координаты угловых точек фигуры. Для однозначного определения расположения фигуры сначала необходимо задать три точки: *g, e, f.* 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 .

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).

# 2. РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ

## 2.1. Анализ вариантов использования программы

Необходимо разработать программу, вычисляющую площадь фигуры методом Монте-Карло. Программа должна предоставлять пользователю возможность выполнения следующих действий:

* ввод координат;
* нахождение площади фигуры методом Монте-Карло;
* очистка полей ввода;
* повторный расчет;
* выход.

На рис.2 представлена диаграмма вариантов использования приложений. В диаграмме были использованы такие виды отношений как расширение, включение и наследование. Расширение означает, что процедура выполняется при определенных условиях. Включение -разновидность отношения зависимости, которое выполняется всегда. Наследование представляет собой связь между родителем и потомком, который наследует его свойства.

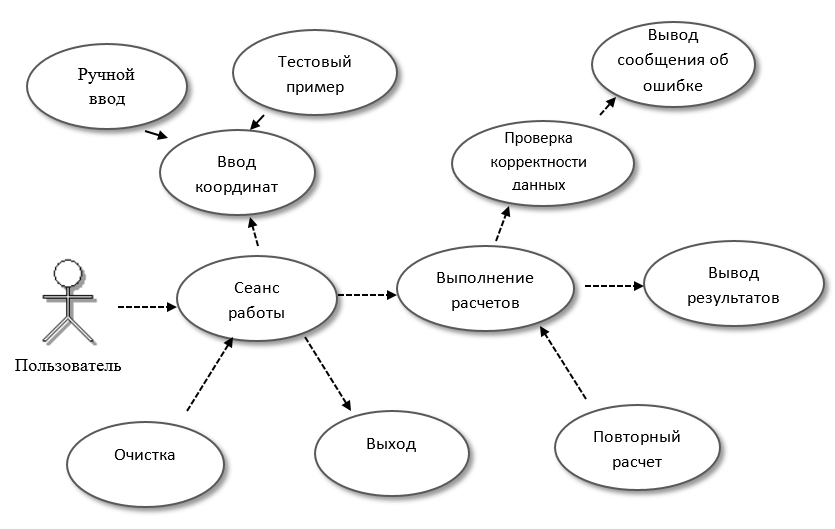


Рис.2. Диаграмма вариантов использования приложений

## 2.2. Требования к приложению

**2.2.1. Общие т**ребования

* Программа должна поддерживать следующие функции: ввод, расчёт, очистка, выход.
* Координаты представляют собой пару значений *x* и *y* для каждой точки фигуры. Они вводятся пользователем с помощью соответствующих текстовых полей и представляют собой значения типа double.
* При вводе значений необходимо проверять корректность вводимых данных и соответствие вершин форме треугольника *gef*.
* После вычисления площади треугольника приложение должно выводить в консоль количество точек *N*, количество попавших в фигуру точек, высчитанную по методу Монте-Карло площадь, погрешность и время работы для .

### 2.2.2. Требования к процедурному приложению

Фигура должна описываться массивом точек, представленного в виде глобальной переменной.

### 2.2.3. Требования к объектно-ориентированному приложению

* Фигура должна описываться предназначенным для этого классом Triangle.
* Все методы, задействованные в поиске площади, должны располагаться в одном классе. Работа с этими методами должна производится через объект этого класса.

# Каждый класс должен иметь три типа конструкторов: по умолчанию, инициализации, копии.3. АНАЛИЗ АЛГОРИТМА РЕШЕНИЯ

Пусть координаты точек заданной фигуры (рис. 3) объявлены следующим образом: *g(xg, yg), e(xe, ye), f(xf, yf)*.

Координаты всех точек задаются пользователем.

## 3.1. Метод Монте-Карло

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

*Sabec = (ex – ax) ∙ (ey – ay)*.

Так как *ax* = *gx*, а *ay* = *fy*, площадь прямоугольника можно записать в следующем виде:

*Sabec = (ex – gx) ∙ (ey – fy)*.

Сгенерируем в прямоугольнике *N* точек. Их координаты высчитываются случайным образом.

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

Приближённую площадь фигуры вычислим по следующей формуле:

*SMK* = *Sabec.*

Для нахождения действительной площади фигуры используется формула Герона:

,

где *p* = , *ge*, *gf*, *ef* – длина сторон треугольника.

Длину стороны можно найти как гипотенузу в прямоугольном треугольнике:

,

где *l* – длина стороны треугольника, и – координаты первой вершины, а и – координаты второй вершины.

Для нахождения погрешности вычислений в процентах методом Монте-Карло используется следующая формула:

.

## 3.2. Определение попадания точки внутрь фигуры

Необходимо сгенерировать внутри прямоугольника *abec* точку *p(xp, yp)* и определить, принадлежит ли эта точка фигуре *gef*, если известны координаты точек фигуры.

Уравнение прямой *gf* имеет вид:

*ygf(x) = kgf ∙ x+bgf*, где , .

Уравнение прямой *ge* имеет вид:

*yge(x) = kge ∙ x +bge*, где , .

Уравнение прямой *ef* имеет вид:

*yef(x) = kef ∙ x +bef*, где , .

Для определения попадания точки внутрь фигуры используется следующий алгоритм:

1. если *xp* < *xf* , то точка принадлежит фигуре, если *yp* ≥ *ygf(xp)* и *yp* ≤ *yge(xp)*;
2. если *xp* > *xf* , то точка принадлежит фигуре, если *yp* ≥ *yef(xp)* и *yp* ≤ *yge(xp)*;
3. если *xp* = *xf* , то точка принадлежит фигуре, если *yp* ≤ *yge(xp)*.

# 4. ПРОЦЕДУРНОЕ ПРИЛОЖЕНИЕ НА БАЗЕ 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::Windows Forms для создания приложений.

## 4.2. Логическое проектирование

Все методы, которые используются для нахождения площади, расположены в основном классе Form1. Этот класс среди глобальных переменных содержит массив точек, благодаря чему каждый метод будет иметь доступ к ним. Точки являются объектами класса Point и содержат поля, описывающие координаты *x* и *y*.

## 4.3. Системные требования

Для корректной работы приложения требуется:

* Операционная система Windows XP 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 Form1 представлена на рис. 3.

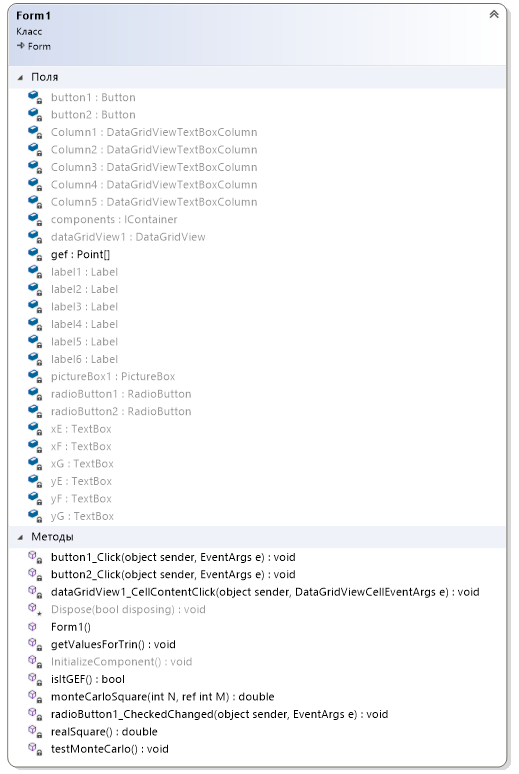


Рис. 3. Диаграмма класса Form1

Приложение содержит следующие глобальные переменные:

– массив вершин фигуры;

– поля ввода данных и кнопки управления.

Приложение cодержит следующие функции:

private void button1\_Click (object sender, EventArgs e) – основной метод, вызывает метод MonteCarlo();;

private void GetValues() – получает координаты трёх точек треугольника из соответствующих текстовых полей и попутно проверяет корректность введённых данных. После происходит проверка с помощью метода IsItTriangle на соответствие формы введённого треугольника необходимой, и если нет, то приложение попросит ввести координаты ещё раз;

private bool IsItTriangle() – проверяет соответствие введённых координат треугольнику *gef*;

private void MonteCarlo() – находит с помощью метода monteCarloSquare площадь и выводит результаты в таблицу. Сначала в этом методе найдётся точная площадь фигуры с помощью метода RealSQR и создастся объект класса Stopwatch для подсчёта времени. Далее будут подсчитаны и выведены в таблицу площадь, время, погрешность и прочие параметры для ;

private double monteCarloSquare(int N, ref int M) – находит площадь фигуры методом Монте-Карло;

private double RealSQR() – находит точную площадь треугольника по формуле Герона;

Более детально алгоритм методов MonteCarlo и monteCarloSquare можно рассмотреть на рис. 4.

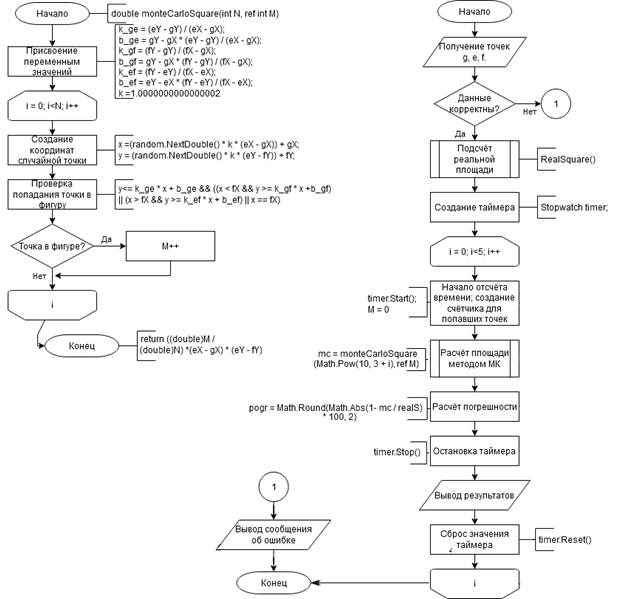


Рис. 4. Схемы методов MonteCarlo и monteCarloSquare класса Form1

## 4.5. Результаты работы

Результаты работы приложения представлены на рис. 5.

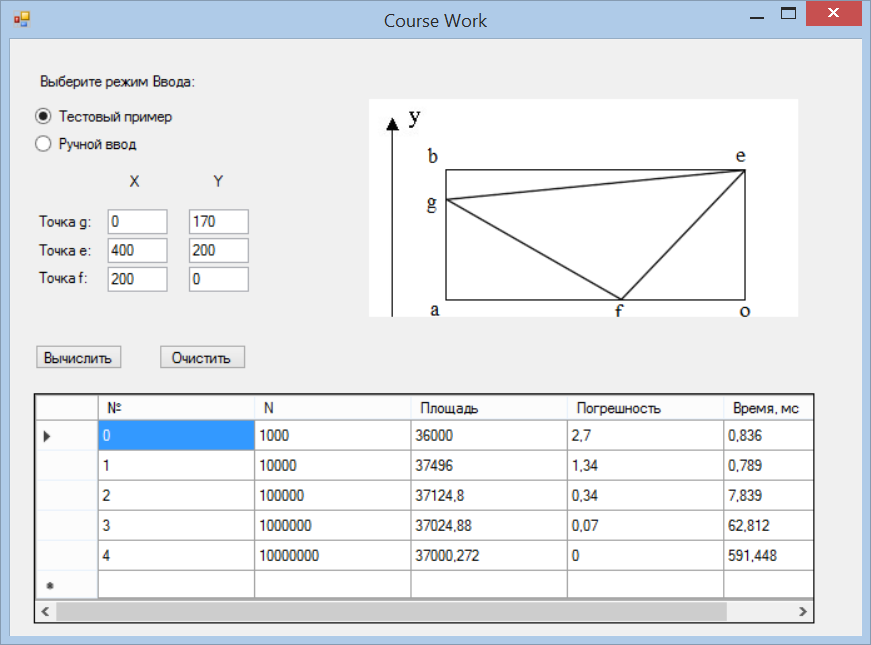


Рис. 5. Тестирование процедурного приложения5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРИЛОЖЕНИЕ НА БАЗЕ WINDOWS FORMS C#

## 5.1. Программные средства разработки

Объектно-ориентированное приложение выполняется в инструментальной системе MS Visual Studio 19. 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::Windows.Forms для создания приложений.

## 5.2. Логическое проектирование

В рассматриваемой предметной области можно выделить следующие сущности: точка, треугольник. Размеры и форма треугольника задаются тремя точками. Каждая точка задаётся двумя координатами: *х* и *у*. Для сущности точки можно создать класс Point, который будет содержать конструкторы и поля *х*, *у*. Для сущности треугольник можно создать класс Triangle, который будет содержать конструкторы и три объекта класса Point. Для удобства все методы, участвующие в нахождении площади методом Монте-Карло, также размещаются в классе Triangle. Диаграмма классов объектно-ориентированного приложения представлена на рис. 6.

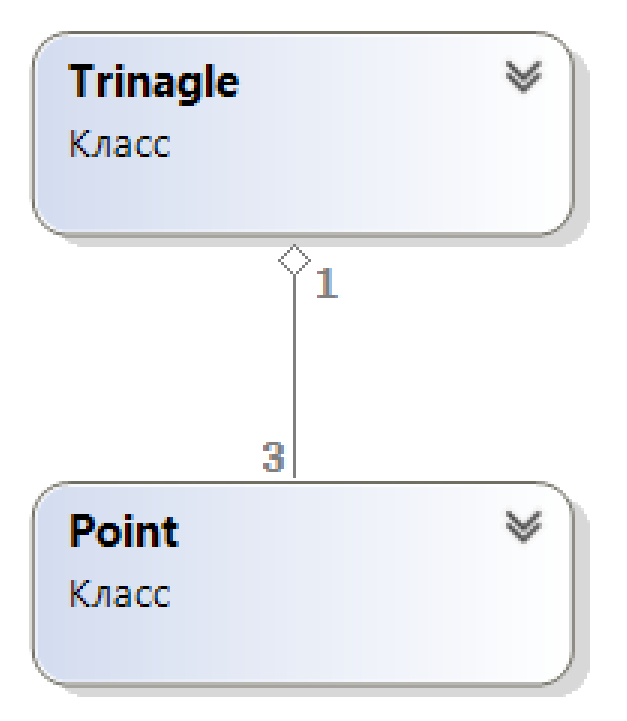


Рис. 6. Диаграмма классов объектно-ориентированного приложения

## 5.3. Описание программы

Класс Point представляет собой точку с двумя координатами. Диаграмма класса представлена на рис. 7.

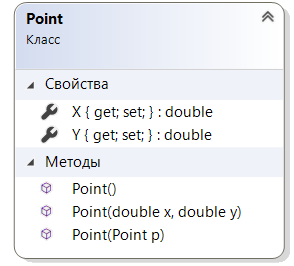


Рис. 7. Диаграмма класса Point

Класс Point содержит следующие свойства:

public double X {get; private set;} – координата *x* точки;

public double Y {get; private set;} – 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 Triangle представляет собой три точки с методами по поиску площади. Диаграмма класса представлена на рис. 8.

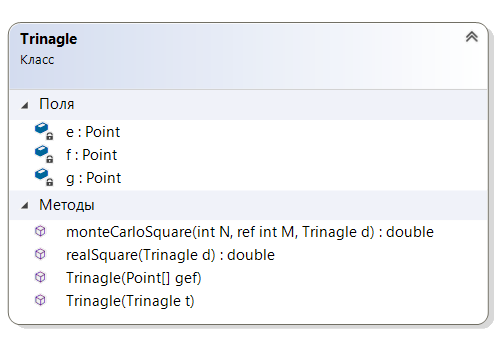


Рис. 8. Диаграмма класса Triangle

Класс Triangle содержит следующие поля:

Point g, f, e – вершины треугольника.

Класс Triangle содержит следующие конструкторы:

public Triangle () – по умолчанию;

public Triangle (Point \_g, Point \_e, Point \_f) – инициализации;

public Triangle (Triangle t) – копии.

Класс Triangle содержит следующие методы:

private void GetValues() – получает координаты трёх точек треугольника из консоли и попутно проверяет корректность введённых данных. После метод проверяет с помощью метода IsItTriangle, соответствует ли форма введённого треугольника необходимой, и если нет, то приложение попросит ввести координаты ещё раз;

private bool IsItTriangle() – проверяет соответствие введённых координат треугольнику *gef*;

private void MonteCarlo() – находит площадь и выводит результаты в консоль. Сначала в этом методе найдётся точная площадь фигуры с помощью метода RealSQR и создастся объект класса Stopwatch для подсчёта времени. Далее будут подсчитаны и выведены в консоль площадь, время, погрешность и прочие параметры для ;

private double monteCarloSquare(int N, ref int M) – Находит площадь треугольника методом Монте-Карло;

private double RealSQR() – находит точную площадь треугольника по формуле Герона.

Класс Form1 вызывает по нажатию кнопки функцию MonteCarlo().

## 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 трёх классов: Form1, Point и Triangle. Класс Form1 содержит метод button1\_Click, который создаёт объект класса Triangle и выполняет над ним определённые действия. Класс Triangle содержит три объекта класса Point и методы, которые отвечают за создание класса и поиск площади методом Монте-Карло. Класс Point содержит координаты точки *x*, *y* и конструкторы.

**5.4.4. Проверка программы**

Для тестирования программы можно выбрать координаты фигуры по умолчанию. В результатах вычисления программы площадь фигуры должна составлять 37000 единиц.

**5.4.5. Сообщения системному программисту**

В ходе выполнения программы возможен вывод следующих сообщений об ошибках:

* Необходимо ввести ровно 3 значения;
* Неверный формат введённых значений;
* Введённые координаты не соответствую треугольнику GEF.

При появлении первой ошибки необходимо повторно ввести координаты точки x и y, при этом количество значений должно равняться двум, иначе ошибка возникнет вновь.

При появлении второй ошибки необходимо повторно ввести координаты точки x и y, при этом вводимые значения должны быть численными и в вещественных числах должен использоваться символ “,”, иначе ошибка возникнет вновь.

При появлении третей ошибки нужно повторно ввести координаты всех точек, так как введённые координаты не соответствую треугольнику *gef*. Чтобы фигура соответствовала *gef*, необходимо выполнение следующих условий: *xg < xf, xf < xe, yg > yf, ye > yg*.

## 5.5. Руководство программиста

**5.5.1. Назначение и условие применения программы**

Приложение разработано для вычисления площади фигуры методом Монте-Карло. Программа поддерживает выполнение следующих функций:

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

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 560 мс.

**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:

* Необходимо ввести ровно два значения;
* Неверный формат введённых значений;
* Введённые координаты не соответствую треугольнику GEF.

При появлении первой ошибки необходимо повторно ввести координаты точки x и y, при этом количество значений должно равняться двум, иначе ошибка возникнет вновь.

При появлении второй ошибки необходимо повторно ввести координаты точки x и y, при этом вводимые значения должны быть численными и в вещественных числах должен использоваться символ “,”, иначе ошибка возникнет вновь.

При появлении третей ошибки нужно повторно ввести координаты всех точек, так как введённые координаты не соответствую треугольнику *gef*. Чтобы фигура соответствовала *gef*, необходимо выполнение следующих условий: *xg < xf, xf < xe, yg > yf, ye > yg*.

## 5.6. Руководство оператора

**5.6.1 Назначение программы**

Приложение разработано для вычисления площади фигуры методом Монте-Карло. Программа поддерживает выполнение следующих функций:

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

**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**

В самом начале и в конце каждого этапа работы программа предлагает пользователю ввести три координаты точки, выбрать ручной ввод или тесовый пример, нажать одну из двух кнопок – Решить или Очистить.

При выборе тестового примера программа берет значения по умолчанию и не считывает их с полей. Иначе – значения считываются с текстовых полей.

При нажатии на кнопку Решить происходит запуск вычислений и вывод результата в таблицу.

При нажатии на кнопку Очистить происходит очищение таблицы.

Выход из приложения осуществляется нажатием кнопки выхода, расположенной в форме по умолчанию в правом верхнем углу.

**5.6.4. Сообщения оператору**

В ходе выполнения программы возможен вывод следующих сообщений об ошибках:

* Необходимо ввести ровно 3 значения;
* Неверный формат введённых значений;
* Введённые координаты не соответствую треугольнику GEF.

При появлении первой ошибки необходимо повторно ввести координаты точки x и y, при этом количество значений должно равняться двум, иначе ошибка возникнет вновь.

При появлении второй ошибки необходимо повторно ввести координаты точки x и y, при этом вводимые значения должны быть численными и в вещественных числах должен использоваться символ “,”, иначе ошибка возникнет вновь.

При появлении третей ошибки нужно повторно ввести координаты всех точек, так как введённые координаты не соответствую треугольнику *gef*. Чтобы фигура соответствовала *gef*, необходимо выполнение следующих условий: *xg < xf, xf < xe, yg > yf, ye > yg*.

## 5.7. Результаты работы

Результаты работы приложения представлены на рис. 9.

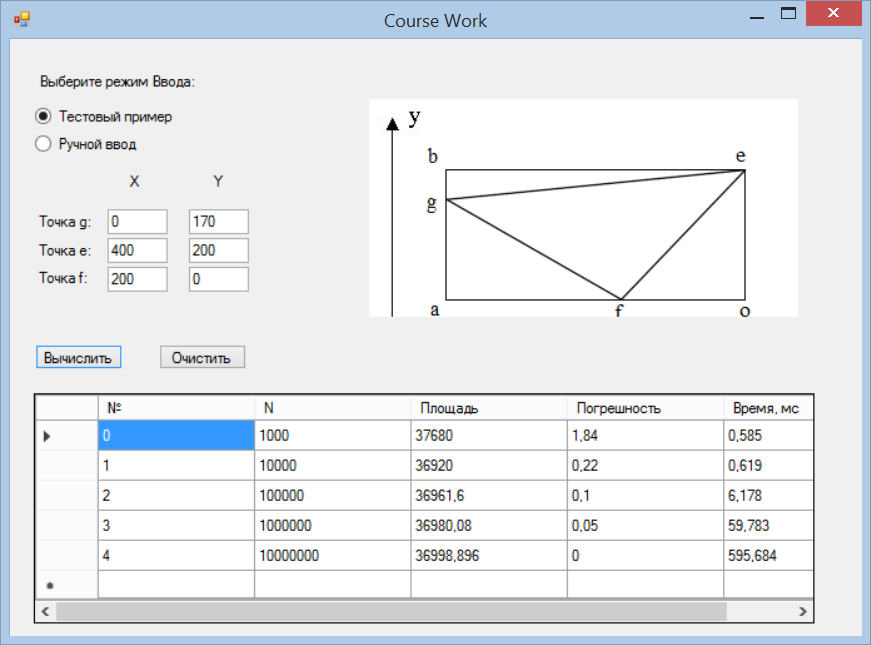


Рис. 9. Тестирование объектно-ориентированного приложения

# 6. АНАЛИЗ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ ПРИЛОЖЕНИЙ

Приложения тестировались на компьютере со следующими характеристиками:

* Операционная система Windows 10;
* Процессор Intel(R) Core(TM) i5-8250U с частотой 1.60 GHz;
* Оперативная память 6 ГБ;
* Видеокарта NVIDIA GeForce GTX 1050.

Результаты сравнения двух приложений представлены в таблице 1.

Таблица 1 – продолжительность вычислений в двух приложениях

|  |  |  |
| --- | --- | --- |
| Количество точек | Время выполнения, мс | |
| Процедурное приложение на базе Windows Worms/C# | Объектно-ориентированное приложение на базе Windows Worms/C# |
|  | 0,671 | 0,585 |
|  | 0,654 | 0,619 |
|  | 6,792 | 6,178 |
|  | 74,89 | 59,783 |
|  | 660,786 | 595,684 |

В среднем оба предложения обладают одинаковой скоростью работы.

# 7. 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. 10) 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. 10), что сильно скажется на производительности программы.

В этом же окне нужно снять галочку с пункта “Предпочтительно 32-рарз” (рис. 10). 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. 10), что в итоге незначительно увеличит производительность.

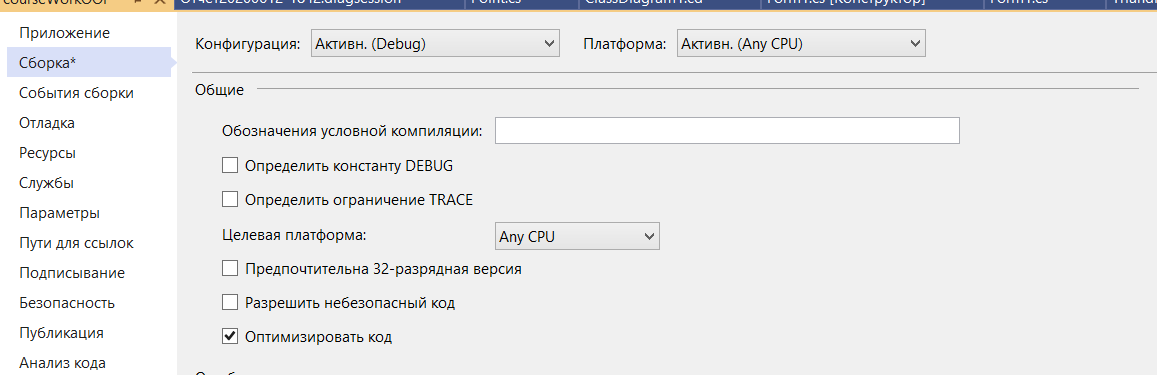


Рис. 10. Вкладка “Сборка” в свойствах проекта

## 7.2. Профилирование

Для приложений было проведено профилирование по использованию ЦП. Результаты приведены на рис. 11 и 12.

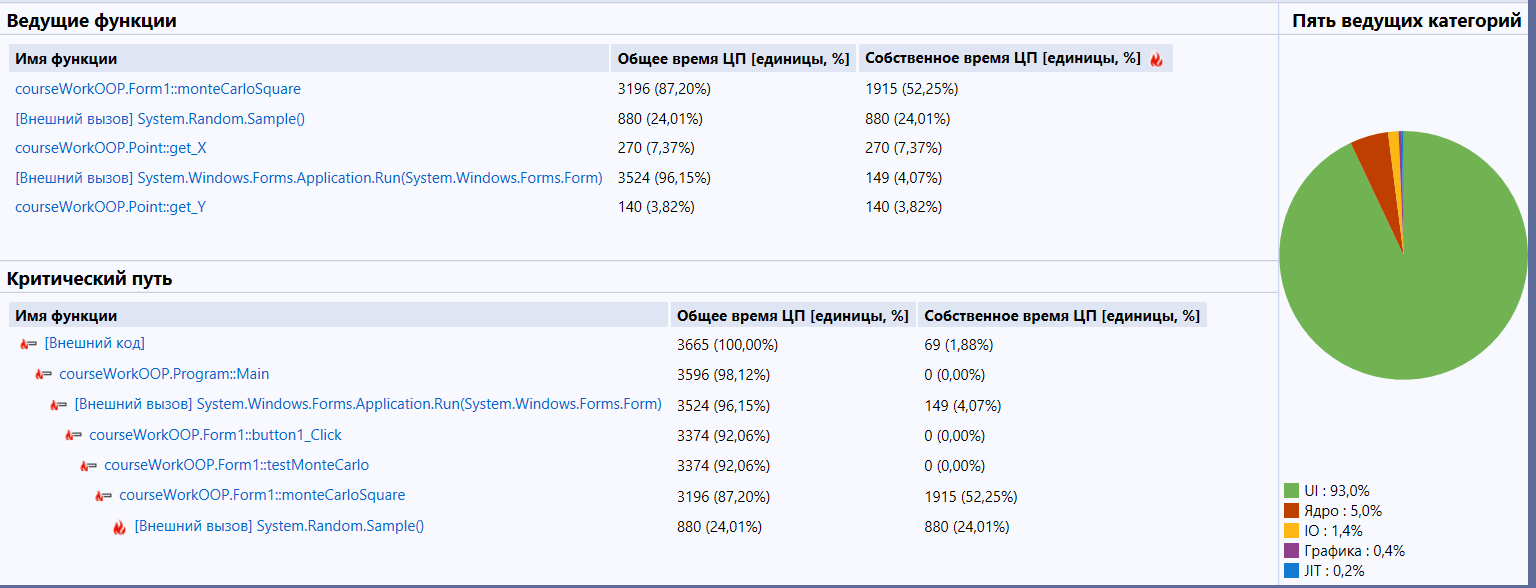
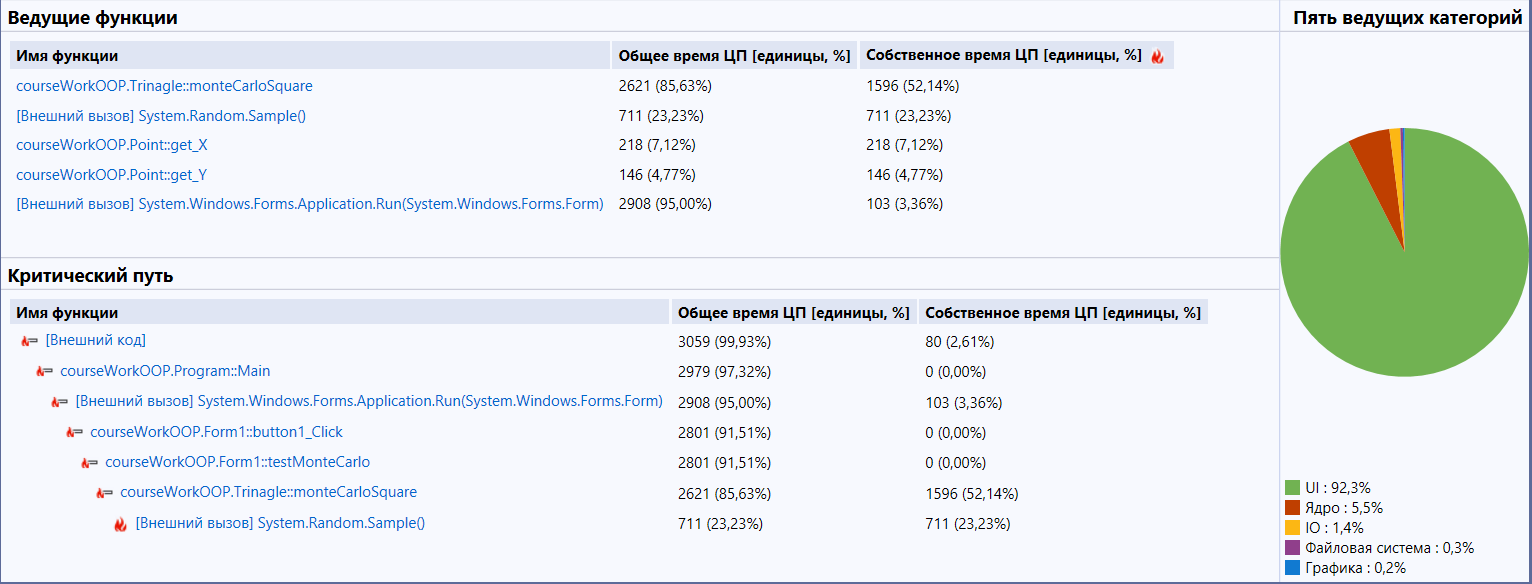


Рис. 11. Результаты профилирования процедурного приложения

Рис. 12. Результаты профилирования объектно-ориентированного приложения

Из результатов видно, что больше всего ресурсов уходит на метод monteCarloSquare. 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, мс | | | |
| Процедурное приложение на базе Windows Forms/C# | | Объектно-ориентированное приложение на базе Windows Forms/C# | |
| До оптим. | После оптим. | До оптим. | После оптим. |
|  | 0,671 | 0,04 | 0,07 | 0,038 |
|  | 0,654 | 0,364 | 0,636 | 0,319 |
|  | 6,792 | 3,446 | 6,273 | 3,156 |
|  | 74,89 | 33,685 | 61,585 | 31,522 |
|  | 660,786 | 261,705 | 647,523 | 239,537 |

В среднем процедурное приложение стало работать на 60,54 % быстрее, а объектно-ориентированное приложение на 71,41% 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 Form1.cs (процедурное приложение)

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Diagnostics;

namespace courseWorkPOP

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

static Point[] gef = new Point[3];

private void button1\_Click(object sender, EventArgs e)

{

MonteCarlo();

}

private void dataGridView1\_CellContentClick(object sender, DataGridViewCellEventArgs e) { }

private static double RealSQR()

{

double ge = Math.Sqrt(Math.Pow(gef[1].X - gef[0].X, 2) + Math.Pow(gef[1].Y - gef[0].Y, 2));

double gf = Math.Sqrt(Math.Pow(gef[2].X - gef[0].X, 2) + Math.Pow(gef[0].Y - gef[2].Y, 2));

double ef = Math.Sqrt(Math.Pow(gef[1].X - gef[2].X, 2) + Math.Pow(gef[1].Y - gef[2].Y, 2));

double p = (ge + gf + ef) / 2;

return Math.Sqrt(p \* (p - ge) \* (p - gf) \* (p - ef));

}

private static double monteCarloSquare(int N, ref int M)

{

double k\_ge = (gef[1].Y - gef[0].Y) / (gef[1].X - gef[0].X);

double b\_ge = gef[0].Y - gef[0].X \* (gef[1].Y - gef[0].Y) / (gef[1].X - gef[0].X);

double k\_gf = (gef[2].Y - gef[0].Y) / (gef[2].X - gef[0].X);

double b\_gf = gef[0].Y - gef[0].X \* (gef[2].Y - gef[0].Y) / (gef[2].X - gef[0].X);

double k\_ef = (gef[2].Y - gef[1].Y) / (gef[2].X - gef[1].X);

double b\_ef = gef[1].Y - gef[1].X \* (gef[2].Y - gef[1].Y) / (gef[2].X - gef[1].X);

Random random = new Random();

double k = 1.0000000000000002;

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

{

double x = (random.NextDouble() \* k \* (gef[1].X - gef[0].X)) + gef[0].X;

double y = (random.NextDouble() \* k \* (gef[1].Y - gef[2].Y)) + gef[2].Y;

if (y <= k\_ge \* x + b\_ge && ((x < gef[2].X && y >= k\_gf \* x + b\_gf) || (x > gef[2].X && y >= k\_ef \* x + b\_ef) || x == gef[2].X))

M++;

}

return ((double)M / (double)N) \* (gef[1].X - gef[0].X) \* (gef[1].Y - gef[2].Y);

}

private static bool IsItTriangle()

{

return gef[0].X < gef[2].X && gef[2].X < gef[1].X && gef[0].Y > gef[2].Y && gef[1].Y > gef[0].Y;

}

private void GetValues()

{

if (radioButton1.Checked == true)

{

gef[0] = new Point(0, 170);

gef[1] = new Point(400, 200);

gef[2] = new Point(200, 0);

}

else

{

double s;

if (double.TryParse(xG.Text, out s)==false && double.TryParse(xE.Text, out s) ==false

&&double.TryParse(xE.Text, out s) ==false

&& double.TryParse(yG.Text, out s) ==false && double.TryParse(xF.Text, out s) ==false

&& double.TryParse(yF.Text, out s)) ==false)

{ MessageBox.Show("Данные введены неверно"); return; }

gef[0] = new Point(double.Parse(xG.Text), double.Parse(xE.Text));

gef[1] = new Point(double.Parse(xE.Text), double.Parse(yG.Text));

gef[2] = new Point(double.Parse(xF.Text), double.Parse(yF.Text));

}

}

private void MonteCarlo()

{

GetValues();

double realS = RealSQR();

Random r = new Random();

Stopwatch timer = new Stopwatch();

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

{

timer.Start();

int M = 0;

double mc = monteCarloSquare((int)Math.Pow(10, 3 + i), ref M);

double pogr = Math.Round(Math.Abs(1 - mc / realS) \* 100, 2);

timer.Stop();

dataGridView1.Rows.Add(i, Math.Pow(10, (3 + i)), mc, pogr, Math.Round(timer.Elapsed.TotalMilliseconds, 3));

timer.Reset();

}

}

private void radioButton1\_CheckedChanged(object sender, EventArgs e)

{

}

private void button2\_Click(object sender, EventArgs e)

{

dataGridView1.Rows.Clear();

}

}

}

## Файл Form1.cs (объектно-ориентированное приложение)

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Diagnostics;

namespace courseWorkOOP

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

static Point[] gef = new Point[3];

private void button1\_Click(object sender, EventArgs e)

{

MonteCarlo();

}

private void dataGridView1\_CellContentClick(object sender, DataGridViewCellEventArgs e) { }

public static bool IsItTriangle()

{

return gef[0].X < gef[2].X && gef[2].X < gef[1].X && gef[0].Y > gef[2].Y && gef[1].Y > gef[0].Y;

}

private void GetValues()

{

if (radioButton1.Checked == true)

{

gef[0] = new Point(0, 170);

gef[1] = new Point(400, 200);

gef[2] = new Point(200, 0);

}

else

{

double s;

if (double.TryParse(xG.Text, out s)==false && double.TryParse(xE.Text, out s) ==false

&&double.TryParse(xE.Text, out s) ==false

&& double.TryParse(yG.Text, out s) ==false && double.TryParse(xF.Text, out s) ==false

&& double.TryParse(yF.Text, out s)) ==false)

{ MessageBox.Show("Данные введены неверно"); return; }

gef[0] = new Point(double.Parse(xG.Text), double.Parse(xE.Text));

gef[1] = new Point(double.Parse(xE.Text), double.Parse(yG.Text));

gef[2] = new Point(double.Parse(xF.Text), double.Parse(yF.Text));

}

}

private void MonteCarlo()

{

GetValues();

Triangle trin = new Triangle(gef);

double realS = Triangle.RealSQR(trin);

Random r = new Random();

Stopwatch timer = new Stopwatch();

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

{

timer.Start();

int M = 0;

double mc = Triangle.monteCarloSquare((int)Math.Pow(10, 3 + i), ref M, trin);

double pogr = Math.Round(Math.Abs(1 - mc / realS) \* 100, 2);

timer.Stop();

dataGridView1.Rows.Add(i, Math.Pow(10, (3 + i)), mc, pogr, Math.Round(timer.Elapsed.TotalMilliseconds, 3));

timer.Reset();

}

}

private void button2\_Click(object sender, EventArgs e)

{

dataGridView1.Rows.Clear();

}

}

}

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

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace courseWorkOOP

{

class Triangle

{

Point g;

Point e;

Point f;

public Triangle(Point[] gef)

{

g = new Point(gef[0]);

e = new Point(gef[1]);

f = new Point(gef[2]);

if (!Form1.IsItTriangle())

{

g = new Point(0, 170);

e = new Point(400, 200);

f = new Point(200, 0);

}

}

public Triangle(Triangle t)

{

g = new Point(g);

e = new Point(e);

f = new Point(f);

}

static public double RealSQR(Triangle d)

{

double ge = Math.Sqrt(Math.Pow(d.e.X - d.g.X, 2) + Math.Pow(d.e.Y - d.g.Y, 2));

double gf = Math.Sqrt(Math.Pow(d.f.X - d.g.X, 2) + Math.Pow(d.g.Y - d.f.Y, 2));

double ef = Math.Sqrt(Math.Pow(d.e.X - d.f.X, 2) + Math.Pow(d.e.Y - d.f.Y, 2));

double p = (ge + gf + ef) / 2;

return Math.Sqrt(p \* (p - ge) \* (p - gf) \* (p - ef));

}

public static double monteCarloSquare(int N, ref int M, Triangle d)

{

double k\_ge = (d.e.Y - d.g.Y) / (d.e.X - d.g.X);

double b\_ge = d.g.Y - d.g.X \* (d.e.Y - d.g.Y) / (d.e.X - d.g.X);

double k\_gf = (d.f.Y - d.g.Y) / (d.f.X - d.g.X);

double b\_gf = d.g.Y - d.g.X \* (d.f.Y - d.g.Y) / (d.f.X - d.g.X);

double k\_ef = (d.f.Y - d.e.Y) / (d.f.X - d.e.X);

double b\_ef = d.e.Y - d.e.X \* (d.f.Y - d.e.Y) / (d.f.X - d.e.X);

Random random = new Random();

double k = 1.0000000000000002;

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

{

double x = (random.NextDouble() \* k \* (d.e.X - d.g.X)) + d.g.X;

double y = (random.NextDouble() \* k \* (d.e.Y - d.f.Y)) + d.f.Y;

if (y <= k\_ge \* x + b\_ge && ((x < d.f.X && y >= k\_gf \* x + b\_gf) || (x > d.f.X && y >= k\_ef \* x + b\_ef) || x == d.f.X))

M++;

}

return ((double)M / (double)N) \* (d.e.X - d.g.X) \* (d.e.Y - d.f.Y);

}

}

}

## Файл Point.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace courseWorkOOP

{

class Point

{

public double X { get; private set; }

public double Y { get; private set; }

public Point()

{

X = 0;

Y = 0;

}

public Point(double x, double y)

{

X = x;

Y = y;

}

public Point(Point p)

{

X = p.X;

Y = p.Y;

}

}

}