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\_\_\_ \_\_\_\_Илюхин Д.М. \_\_\_\_\_\_\_\_\_\_\_

(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 Илюхину Дмитрию Михайловичу

Тема работы:

«Исследование вычислительной эффективности объектно- ориентированных приложений»

Входные данные Вариант №24:

задача: определение площади методом Монте-Карло;

фигура: gef;

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](#_Toc42529201)

[1. ПОСТАНОВКА ЗАДАЧИ 6](#_Toc42529202)

[2. РАЗРАБОТКА ТЕХНИЧЕСКОГО ЗАДАНИЯ 8](#_Toc42529203)

[2.1. Анализ вариантов использования программы 8](#_Toc42529204)

[2.2. Требования к приложению 9](#_Toc42529205)

[3. АНАЛИЗ АЛГОРИТМА РЕШЕНИЯ 11](#_Toc42529206)

[3.1. Метод Монте-Карло 11](#_Toc42529207)

[3.2. Определение попадания точки внутрь фигуры 12](#_Toc42529208)

[4. ПРОЦЕДУРНОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C# 13](#_Toc42529209)

[4.1. Программные средства разработки 13](#_Toc42529210)

[4.2. Логическое проектирование 13](#_Toc42529211)

[4.3. Системные требования 15](#_Toc42529212)

[4.4. Структура приложения 15](#_Toc42529213)

[4.5. Результаты работы 16](#_Toc42529214)

[5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРИЛОЖЕНИЕ НА БАЗЕ Console/C# 17](#_Toc42529215)

[5.1. Программные средства разработки 17](#_Toc42529216)

[5.2. Логическое проектирование 17](#_Toc42529217)

[5.3. Описание программы 20](#_Toc42529218)

[5.4. Руководство системного программиста 21](#_Toc42529219)

[5.5. Руководство программиста 23](#_Toc42529220)

[5.6. Руководство оператора 24](#_Toc42529221)

[5.7. Результаты работы 26](#_Toc42529222)

[6. АНАЛИЗ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ ПРИЛОЖЕНИЙ 27](#_Toc42529223)

[7. УЛУЧШЕНИЕ ХАРАКТЕРИСТИК КАЧЕСТВА ПРИЛОЖЕНИЙ 28](#_Toc42529224)

[7.1. Использование встроенной оптимизации кода 28](#_Toc42529225)

[7.2. Профилирование 28](#_Toc42529226)

[7.3. Результаты улучшения характеристик 30](#_Toc42529227)

[ЗАКЛЮЧЕНИЕ 31](#_Toc42529228)

[СПИСОК ЛИТЕРАТУРЫ 32](#_Toc42529229)

[ПРИЛОЖЕНИЕ 34](#_Toc42529230)

[Файл Program.cs (процедурное приложение)](#_Toc42529231)

[Файл Point.cs](#_Toc42529232)

[Файл Program.cs (объектно-ориентированное приложение)](#_Toc42529233)

[Файл Point.cs](#_Toc42529234)

[Файл Trinagle.cs](#_Toc42529235)

# 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 №24:

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

Геометрическая фигура представляет собой обычный треугольник с вершинами *g*, *e* и *f* (рис. 1).

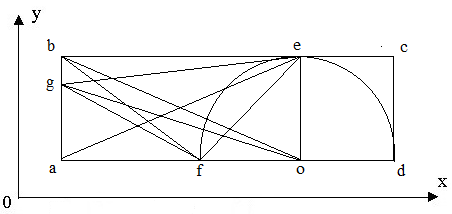


Рис. 1. Фигура *gef*

Исходными данными для каждого приложения являются координаты вершин треугольника. Для однозначного определения расположения фигуры 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 *S ∙ M / N*.

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 треугольника *gef*.
* 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**

Процедурное приложение должно соответствовать следующим требованиям:

* используется процедурная парадигма программирования;
* фигура должна описываться с помощью массива точек, представленного в виде глобальной переменной.

**2.2.3. Требования к объектно-ориентированному приложению**

Объектно-ориентированное приложение должно соответствовать следующим требованиям:

* используется объектно-ориентированная парадигма программирования;
* фигура должна описываться с помощью предназначенного для этого класса (Trinagle).
* все методы, участвующие 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.

# 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*, то площадь прямоугольника можно записать в виде:

*Sabec = (ex – gx) ∙ (ey – fy)*.

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 фигуры используется формула Герона:

,

где *p* – полупериметр; *ge*, *gf*, *ef* – длина сторон треугольника:

*p* = ;

;

*;*

*.*

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

.

## 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. Пример расположения точек фигуры и точки *p* представлен на рис. 3.

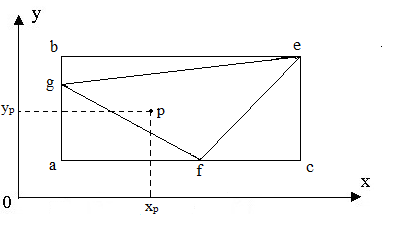


Рис. 3. Точки, необходимые для расчётов

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)*.

# 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. 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*.

Диаграмма класса Program представлена на рис. 4.

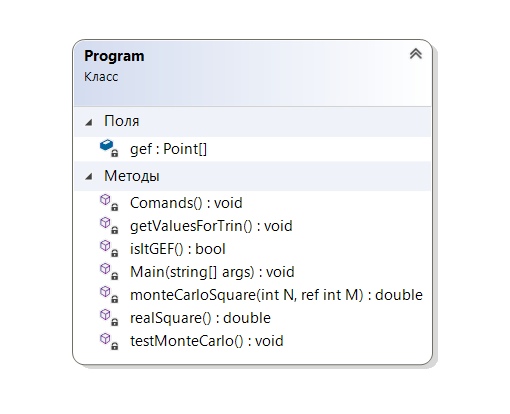


Рис. 4. Диаграмма класса Program

Приложение содержит следующие глобальные переменные:

Point[] gef – массив вершин фигуры.

Приложение одержит следующие функции:

public void Main(string[] args) – основной метод, вызывает метод Comands;

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

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

private void testMonteCarlo() – находит с помощью метода monteCarloSquare площадь и выводит результаты в консоль. Сначала в этом методе найдётся точная площадь фигуры с помощью метода realSquare 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 фигуры методом Монте-Карло;

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

private void Comands() – вызывается в самом начале и в конце каждого этапа для обработки команд пользователя.

Более детально алгоритм методов testMonteCarlo и monteCarloSquare можно рассмотреть на рис. 5.

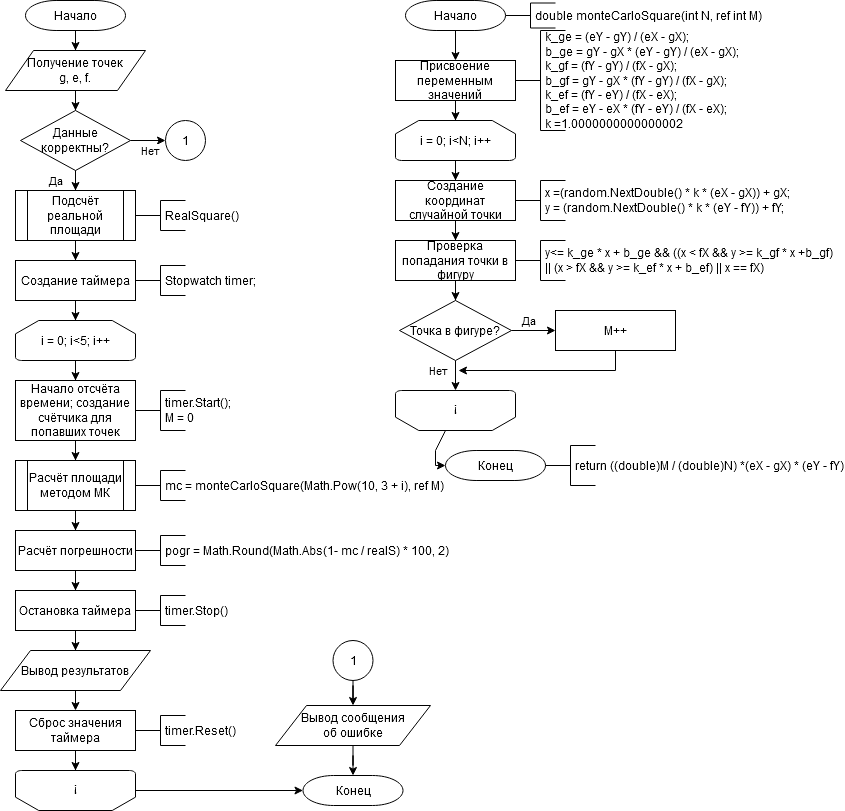


Рис. 5. Схемы методов testMonteCarlo и monteCarloSquare класса Program

## 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

Структура проекта представлена на рисунке 6.

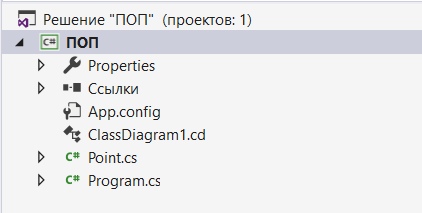


Рис. 6. Проект процедурного приложения

Процедурное приложение включает следующие файлы:

Properties – содержит файлы, определяющие свойства приложения;

Ссылки – содержит ссылки на используемые библиотеки;

App.config – xml файл, определяющий настройки приложения;

ClassDiagram1.cd – файл, содержащий диаграмму классов;

Point.cs – файл, содержащий класс для определения точки;

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

## 4.5. Результаты работы

Результаты работы приложения представлены на рис. 7.

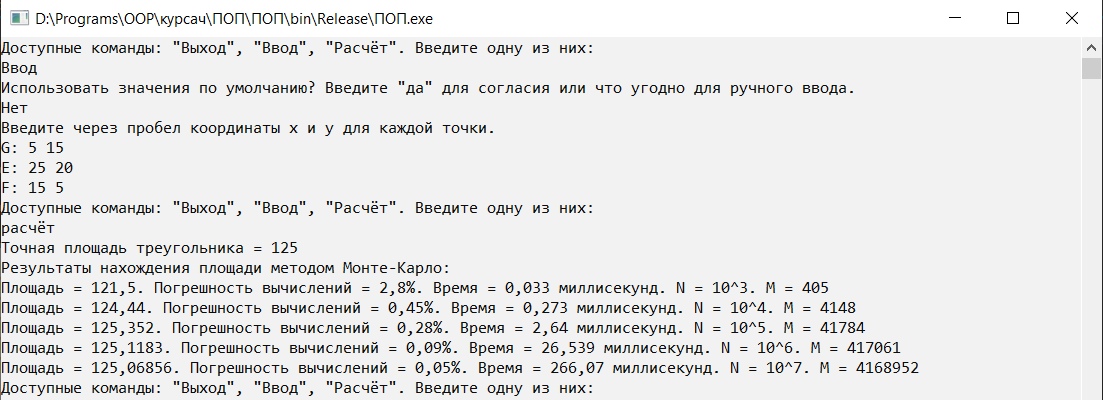


Рис. 7. Тестирование процедурного приложения

# 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, 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 Trinagle, который будет содержать конструкторы и три объекта класса Point. Для удобства все методы, участвующие в нахождении площади методом Монте-Карло, также размещаются в классе Trinagle. Диаграмма классов объектно-ориентированного приложения представлена на рис. 8.

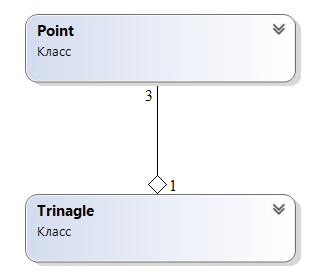


Рис. 8. Диаграмма классов объектно-ориентированного приложения

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

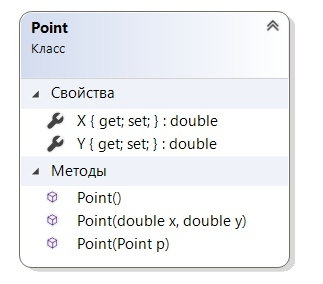


Рис. 9. Диаграмма класса Point

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 Trinagle представляет собой три точки с методами по поиску площади. Диаграмма класса представлена на рис. 10.

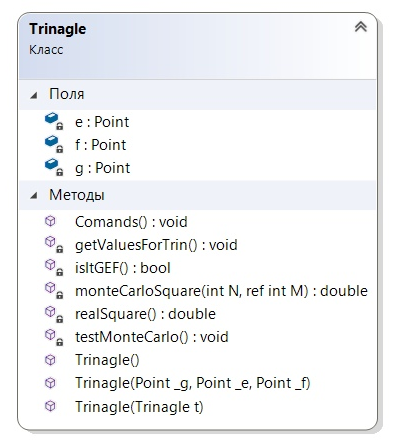


Рис. 10. Диаграмма класса Trinagle

Класс Trinagle содержит следующие поля:

Point g, f, e – вершины треугольника.

Класс Trinagle содержит следующие конструкторы:

public Trinagle () – по умолчанию;

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

public Trinagle (Trinagle t) – копии.

Класс Trinagle содержит следующие методы:

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

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

private void testMonteCarlo() – находит площадь и выводит результаты в консоль. Сначала в этом методе найдётся точная площадь фигуры с помощью метода realSquare 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 realSquare() – находит точную площадь треугольника по формуле Герона;

public void Comands() – вызывается в конце каждого этапа для обработки команд пользователя.

## 5.3. Описание программы

**5.3.1. Общие сведения**

Объектно-ориентированное приложение реализовано в качестве консольного на языке программирования C# с использованием объектно-ориентированного подхода. Для работы приложения требуется .NET Framework 4.5.2 и выше.

**5.3.2. Функциональное назначение**

Приложение разработано для вычисления площади фигуры методом Монте-Карло и имеет следующие возможности:

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

**5.3.3. Описание логической структуры**

Для определения площади фигуры необходимо вначале вычислить площадь прямоугольника, описывающего заданную фигуру, а затем N раз координаты, определяющие точку внутри прямоугольника.

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 *S ∙ M / N*.

Для сущности точки используется класс Point, который будет содержать конструкторы и поля *х*, *у*. Для сущности треугольник используется класс Trinagle, который будет содержать конструкторы и три объекта класса Point. Для удобства все методы, участвующие в нахождении площади методом Монте-Карло, также размещаются в классе Trinagle.

**5.3.4. Входные и выходные данные**

Входными данными являются координаты угловых точек фигуры.

Выходными данными является таблица, содержащая:

* количество точек, генерируемых и участвующих в расчётах;
* количество точек, лежащих в фигуре ;
* настоящая площадь фигуры;
* площадь фигуры, вычисленная методом Монте-Карло;
* относительная погрешность вычисления площади;

время выполнения расчётов.

## 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: Program, Point и Trinagle. Класс Program содержит главный метод Main, который создаёт объект класса Trinagle и выполняет над ним определённые действия. Класс Trinagle 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 125 единиц.

**5.4.5. Сообщения системному программисту**

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*.

## 5.5. 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 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:

* 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 команды: ввод, расчёт, выход.

При вводе команды “ввод” программа предложит выбрать данные по умолчанию. Чтобы согласиться, нужно ввести команду “да”. Чтобы отказаться, нужно ввести любую другую строку. При отказе от данных по умолчанию пользователю необходимо ввести координаты *x* и *y* для каждой точки фигуры.

При вводе команды “расчёт” программа выполнит поиск площади методом Монте-Карло с разным количеством точек несколько раз и выведет результаты в консоль. Если вершинам не заданы координаты, то выполнится команда “ввод”.

При вводе команды “выход” приложение будет закрыто.

**5.6.4. Сообщения оператору**

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*.

## 5.7. Результаты работы

Результаты работы приложения представлены на рис. 11.

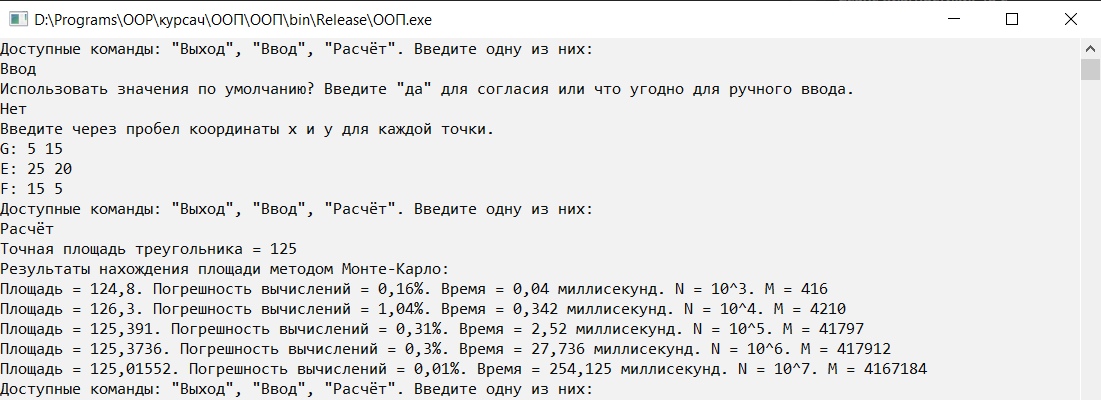


Рис. 11. Тестирование объектно-ориентированного приложения

# АНАЛИЗ ВЫЧИСЛИТЕЛЬНОЙ ЭФФЕКТИВНОСТИ ПРИЛОЖЕНИЙ

Анализ вычислительной эффективности приложений проводился путём сравнения времени расчёта площади фигуры методом Монте-Карло при разном количестве генерируемых точек у каждого приложения.

Процедурное и объектно-ориентированное приложения работали с одними и теми же входными данными. В результате работы каждое приложение осуществляло вычисление с различным количеством точек и выводило полученные данные на экран. Результаты сравнения двух приложений по скорости работы представлены в таблице 1.

Таблица 1 – продолжительность вычислений в двух приложениях

|  |  |  |
| --- | --- | --- |
| Количество точек | Время выполнения, мс | |
| Процедурное приложение на базе Console/C# | Объектно-ориентированное приложение на базе Console/C# |
|  | 0,033 | 0,04 |
|  | 0,273 | 0,342 |
|  | 2,64 | 2,52 |
|  | 26,539 | 27,736 |
|  | 266,07 | 254,125 |

В среднем оба предложения демонстрируют приблизительно одинаковую скорость работы.

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.

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

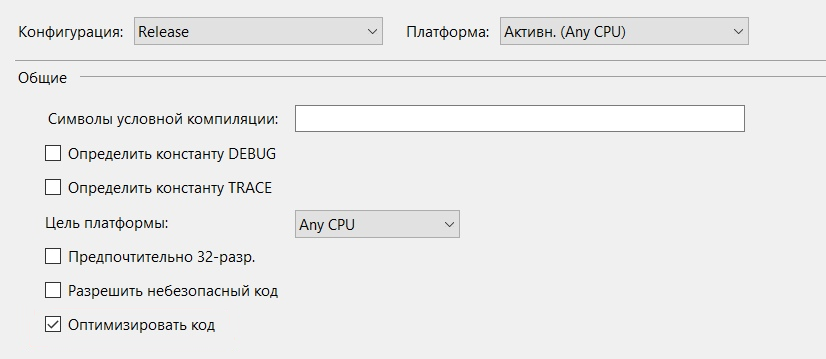


Рис. 12. Вкладка “Сборка” в свойствах проекта

## 7.2. Профилирование

Средства профилирования Visual Studio используются для анализа проблем с производительностью в приложении. Выборка представляет собой метод профилирования, показывающий методы, которые выполняют основную часть работы в пользовательском режиме приложения.

Метод выборки собирает сведения о методах, которые выполняются в приложении, через заданные интервалы времени. После завершения сеанса профилирования представление «Сводка» показывает дерево вызовов наиболее активных методов, которое называется горячим или критическим путем, где была выполнена основная часть работы в приложении. Кроме того представлен график временной шкалы, который демонстрирует нагрузку на систему в разные моменты времени.

Для приложений было проведено профилирование по использованию ЦП. Результаты приведены на рис. 13 и 14.

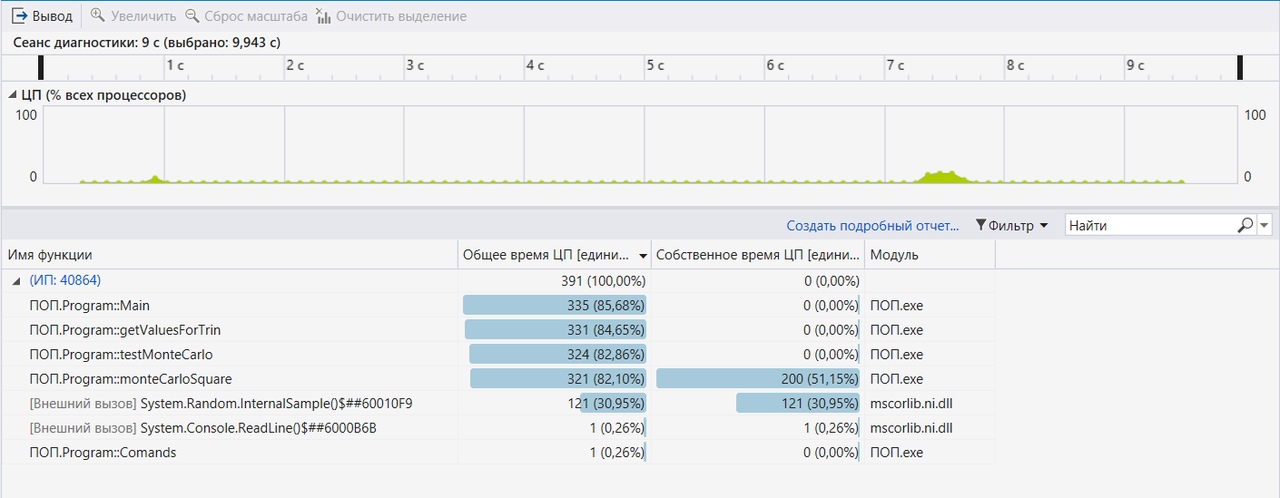


Рис. 13. Результаты профилирования процедурного приложения

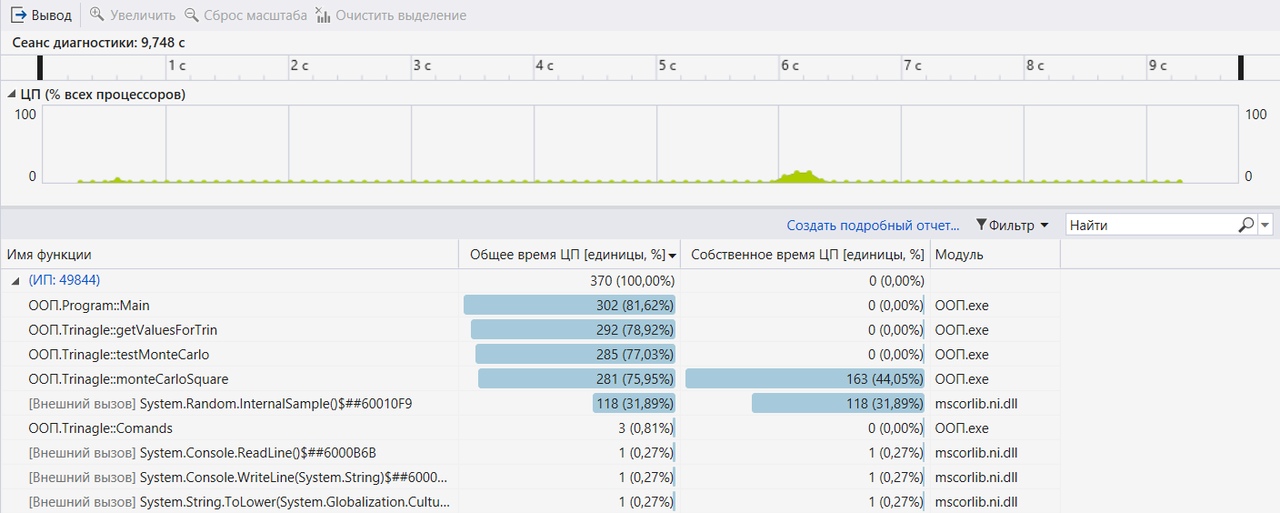


Рис. 14. Результаты профилирования объектно-ориентированного приложения

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 Console/C# | | Объектно-ориентированное приложение на базе Console/C# | |
| До оптим. | После оптим. | До оптим. | После оптим. |
|  | 0,108 | 0,033 | 0,07 | 0,04 |
|  | 0,703 | 0,273 | 0,636 | 0,342 |
|  | 6,609 | 2,64 | 6,273 | 2,52 |
|  | 62,765 | 26,539 | 61,585 | 27,736 |
|  | 625,833 | 266,07 | 647,523 | 254,125 |

В среднем процедурное приложение стало работать на 57 % быстрее, а объектно-ориентированное приложение на 60% 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.
6. ГОСТ 19.701 - 90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.
7. ГОСТ 19.101 - 77 ЕСПД. Виды программ и программных документов.
8. ГОСТ 19.106 - 78 ЕСПД. Требования к программным документам, выполненным печатным способом.
9. ГОСТ 19.401 - 78 ЕСПД. Текст программы. Требования к содержанию и оформлению.
10. ГОСТ 19.402 - 78 ЕСПД. Описание программы.
11. ГОСТ 19.404 - 79 ЕСПД. Пояснительная записка. Требования к содержанию и оформлению.
12. ГОСТ 19.504-79 Руководство программиста. Требования к содержанию и оформлению.
13. ГОСТ 28388-89 СОИ. Документы на магнитных носителях данных. Порядок выполнения и обращения.
14. ГОСТ 51188-98. Испытания программных средств на наличие компьютерных вирусов.
15. 15. ГОСТ 19.503-79. Руководство системного программиста. Требования к содержанию и оформлению.
16. 11. ГОСТ 19.505-79. Руководство оператора. Требования к содержанию и оформлению.

# ПРИЛОЖЕНИЕ

## Файл Program.cs (процедурное приложение)

using System;

using System.Diagnostics;

namespace ПОП

{

class Program

{

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

static void Main(string[] args)

{

Comands();

}

private static double realSquare()

{

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 isItGEF()

{

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 static void getValuesForTrin()

{

Console.WriteLine("Использовать значения по умолчанию? Введите \"да\" для согласия или что угодно для ручного ввода.");

string comand = Console.ReadLine().ToLower();

if(comand == "да")

{

gef[0] = new Point(5, 15);

gef[1] = new Point(25, 20);

gef[2] = new Point(15, 5);

Comands();

return;

}

bool first\_input = true;

string s = "GEF";

while (!isItGEF() || first\_input)

{

Console.WriteLine("Введите через пробел координаты x и y для каждой точки.");

for (int i = 0; i < 3; i++)

{

bool flag = true;

while (flag)

{

Console.Write(s[i]+": ");

string[] coord = Console.ReadLine().Split();

if (coord.Length != 2)

{

Console.WriteLine("Необходимо ввести ровно два значения.");

continue;

}

try

{

gef[i] = new Point(Convert.ToDouble(coord[0]), Convert.ToDouble(coord[1]));

}

catch

{

Console.WriteLine("Неверный формат введённых значений.");

continue;

}

flag = false;

}

}

if (!isItGEF())

Console.WriteLine("Введённые координаты не соответствую треугольнику GEF.");

first\_input = false;

}

Comands();

}

private static void testMonteCarlo()

{

double realS = realSquare();

Random r = new Random();

Stopwatch timer = new Stopwatch();

Console.WriteLine("Точная площадь треугольника = " + realS);

Console.WriteLine("Результаты нахождения площади методом Монте-Карло:");

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();

Console.WriteLine("Площадь = " + mc + ". Погрешность вычислений = " + pogr + "%. Время = " + Math.Round(timer.Elapsed.TotalMilliseconds, 3) + " миллисекунд. N = 10^" + (3 + i) + ". M = "+M);

timer.Reset();

}

Comands();

}

private static void Comands()

{

Console.WriteLine("Доступные команды: \"Выход\", \"Ввод\", \"Расчёт\". Введите одну из них: ");

string comand = Console.ReadLine().ToLower();

if (comand == "выход")

Environment.Exit(0);

else if (comand == "ввод")

getValuesForTrin();

else if (comand == "расчёт")

{

if (gef[2] == null)

{

Console.WriteLine("Отсутствуют координаты вершин.");

getValuesForTrin();

}

else

testMonteCarlo();

}

else

{

Console.WriteLine("Введённой команды не существует, введите другую.");

Comands();

}

}

}

}

## Файл Point.cs

namespace ПОП

{

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;

}

}

}

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

namespace ООП

{

class Program

{

static void Main(string[] args)

{

Trinagle trin = new Trinagle();

}

}

}

## Файл Point.cs

namespace ООП

{

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;

}

}

}

## Файл Trinagle.cs

using System;

using System.Diagnostics;

namespace ООП

{

class Trinagle

{

Point g;

Point e;

Point f;

public Trinagle()

{

Comands();

}

public Trinagle(Point \_g, Point \_e, Point \_f)

{

g = new Point(\_g);

e = new Point(\_e);

f = new Point(\_f);

if (!isItGEF())

{

g = new Point(5, 15);

e = new Point(25, 20);

f = new Point(15, 5);

}

}

public Trinagle(Trinagle t)

{

g = new Point(t.g);

e = new Point(t.e);

f = new Point(t.f);

}

private double realSquare()

{

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

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

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

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

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

}

private double monteCarloSquare(int N, ref int M)

{

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

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

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

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

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

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

Random random = new Random();

double k = 1.0000000000000002;

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

{

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

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

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

M++;

}

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

}

private bool isItGEF()

{

return g != null && e != null && f != null && g.X < f.X && f.X < e.X && g.Y > f.Y && e.Y > g.Y;

}

private void getValuesForTrin()

{

Console.WriteLine("Использовать значения по умолчанию? Введите \"да\" для согласия или что угодно для ручного ввода.");

string comand = Console.ReadLine().ToLower();

if (comand == "да")

{

g = new Point(5, 15);

e = new Point(25, 20);

f = new Point(15, 5);

Comands();

return;

}

bool first\_input = true;

string s = "GEF";

while (first\_input || !isItGEF())

{

Console.WriteLine("Введите через пробел координаты x и y для каждой точки.");

for (int i = 0; i < 3; i++)

{

bool flag = true;

while (flag)

{

Console.Write(s[i] + ": ");

string[] coord = Console.ReadLine().Split();

if (coord.Length != 2)

{

Console.WriteLine("Необходимо ввести ровно два значения.");

continue;

}

try

{

switch (i)

{

case 0:

g = new Point(Convert.ToDouble(coord[0]), Convert.ToDouble(coord[1]));

break;

case 1:

e = new Point(Convert.ToDouble(coord[0]), Convert.ToDouble(coord[1]));

break;

case 2:

f = new Point(Convert.ToDouble(coord[0]), Convert.ToDouble(coord[1]));

break;

}

}

catch

{

Console.WriteLine("Неверный формат введённых значений.");

continue;

}

flag = false;

}

}

if (!isItGEF())

Console.WriteLine("Введённые координаты не соответствую треугольнику GEF.");

first\_input = false;

}

Comands();

}

private void testMonteCarlo()

{

double realS = realSquare();

Random r = new Random();

Stopwatch timer = new Stopwatch();

Console.WriteLine("Точная площадь треугольника = " + realS);

Console.WriteLine("Результаты нахождения площади методом Монте-Карло:");

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();

Console.WriteLine("Площадь = " + mc + ". Погрешность вычислений = " + pogr + "%. Время = " + Math.Round(timer.Elapsed.TotalMilliseconds, 3) + " миллисекунд. N = 10^" + (3 + i) + ". M = " + M);

timer.Reset();

}

Comands();

}

public void Comands()

{

Console.WriteLine("Доступные команды: \"Выход\", \"Ввод\", \"Расчёт\". Введите одну из них: ");

string comand = Console.ReadLine().ToLower();

if (comand == "выход")

Environment.Exit(0);

else if (comand == "ввод")

getValuesForTrin();

else if (comand == "расчёт")

{

if (f == null)

{

Console.WriteLine("Отсутствуют координаты вершин.");

getValuesForTrin();

}

else

testMonteCarlo();

}

else

{

Console.WriteLine("Введённой команды не существует, введите другую.");

Comands();

}

}

}

}