**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационной безопасности**

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

**к курсовой работе**

**по дисциплине «Объектно-ориентированное программирование»**

Тема: **«Разработка приложения на основе принципов объектно-ориентированного подхода»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3363 |  | Минко Д. А. |
| Преподаватель |  | Новакова Н. Е. |

Санкт-Петербург

2024

**ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ**

Студент: Минко Д. А.

Группа: 3363

Тема работы: Разработка приложения на основе принципов объектно-ориентированного подхода

Исходные данные:

Реализовать структуры данных согласно заданию, реализовать вывод

результатов работы.

Содержание пояснительной записки:

Титульный лист, задание на курсовую работу, аннотация, содержание, введение, цель работы, разделы, заключение, список использованной литературы.

Содержание разделов:

Формулировка задания, теоретический аспект задачи, формализация задачи, спецификация классов, руководство оператора, руководство программиста, контрольный пример, листинг программы, выводы.

Предполагаемый объем пояснительной записки:

Не менее 40 страниц.

Дата выдачи задания:

Дата сдачи курсовой работы:

Дата защиты курсовой работы:

|  |  |  |
| --- | --- | --- |
| Студент гр. 3363 |  | Минко Д. А. |
| Преподаватель |  | Новакова Н. Е. |

**АННОТАЦИЯ**

Данная работа содержит в себе решение трех задач на основе объектно-ориентированного подхода. В первой задаче рассмотрены механизмы наследования и иерархия классов, во второй – работа с графами, в третьей – разработка имитационной модели.

На основе этих моделей были разработаны приложения, включающие в себя различные пользовательские интерфейсы. Полученные результаты приведены в работе.

**SUMMARY**

This work contains the solution of three problems based on an object-oriented approach. In the first task, inheritance mechanisms and class hierarchy are considered, in the second - working with graphs, in the third - developing a simulation model.

Based on these models, applications have been developed that include various user interfaces. The results obtained are presented in the work.

СОДЕРЖАНИЕ

[**АННОТАЦИЯ 3**](#_Toc154538407)

[**ВВЕДЕНИЕ 5**](#_Toc154538408)

[**ЦЕЛЬ РАБОТЫ 5**](#_Toc154538409)

[**1. ПЕРВЫЙ РАЗДЕЛ 6**](#_Toc154538410)

[**1.1. Формулировка задания 6**](#_Toc154538411)

[**1.2. Теоретический аспект задачи 6**](#_Toc154538412)

[**1.3. Формализация задачи 6**](#_Toc154538413)

[**1.4. Спецификация классов 10**](#_Toc154538414)

[**1.5. Руководство оператора 13**](#_Toc154538415)

[**1.6. Руководство программиста 13**](#_Toc154538416)

[**1.7. Контрольный пример 14**](#_Toc154538417)

[**1.8. Листинг программы 15**](#_Toc154538418)

[**2. ВТОРОЙ РАЗДЕЛ 19**](#_Toc154538419)

[**2.1. Формулировка задания 19**](#_Toc154538420)

[**2.2. Теоретический аспект задачи 19**](#_Toc154538421)

[**2.3. Формализация задачи 20**](#_Toc154538422)

[**2.4. Спецификация классов 24**](#_Toc154538423)

[**2.5. Руководство оператора 28**](#_Toc154538424)

[**2.6. Руководство программиста 30**](#_Toc154538425)

[**2.7. Контрольный пример 30**](#_Toc154538426)

[**2.8. Листинг программы 32**](#_Toc154538427)

[**ЗАКЛЮЧЕНИЕ 43**](#_Toc154538428)

[**СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 44**](#_Toc154538429)

# ВВЕДЕНИЕ

Курсовая работа направлена на создание приложений на основе объектно-ориентированного подхода на языке C#. В ней рассматриваются иерархии классов и наследования, работа с графами, а также имитационные модели. Курсовая работа состоит из 3 разделов.

# ЦЕЛЬ РАБОТЫ

Целью работы является закрепление теоретических знаний и получение практических навыков разработки программного обеспечения на основе объектно-ориентированного подхода.

Формулировка исходного задания:

* Первый раздел: разработать программу для обеспечения продажи туров.
* Второй раздел: для заданного двудольного графа найти число совершенных паросочетаний P и одно из наибольших паросочетаний. Задачу решить в общем виде. В качестве контрольного примера использовать задание своего варианта.

Изображение выглядит как линия, треугольник, оригами

Автоматически созданное описание

Рисунок 1 – Задание в качестве контрольного варианта

* Третий раздел: требуется создать компьютерную модель обслуживания потока заявок на разгрузку, поступающих от грузовых судов (сухогрузов и танкеров), прибывающих в морской порт. Грузовые суда прибывают в порт согласно расписанию, но возможны опоздания и досрочные прибытия. Расписание включает день и время прибытия, название судна, вид груза и его вес, а также планируемый срок стоянки в порту для разгрузки.

Для разгрузки судов в порту используются три вида разгрузочных кранов, соответствующих трем видам грузов: сыпучим и жидким грузам, контейнерам. Число разгрузочных кранов каждого вида ограничено, так что поступающие заявки на разгрузку одного вида груза образуют очередь. Длительность разгрузки судна зависит от вида и веса его груза, а также некоторых других факторов, например, погодных условий. Любой дополнительный (сверх запланированного срока) день стояния судна в порту (из-за ожидания разгрузки в очереди или из-за задержки самой разгрузки) влечет за собой выплату штрафа (например, 2 тыс. у.е. за каждый дополнительный день простоя судна).

При моделировании прибытия судов отклонение их от расписания рассматривается как случайная величина с равномерным распределением в некотором интервале (например, от -2 до 9 дней). Еще одной случайной величиной, изменяющейся в фиксированном диапазоне (например, от 0 до 12 дней), является время задержки окончания разгрузки судна по сравнению с обычным (зависящим только от вида груза и его веса).

Цель моделирования работы морского порта – определение для заданного расписания прибытия судов минимально достаточного числа кранов в порту, позволяющего уменьшить штрафные суммы. Период моделирования – месяц, шаг моделирования – 1-3 дня. В параметры моделирования следует включить расписание прибытия судов, количество кранов каждого вида, диапазоны разброса случайных величин (отклонения от расписания прибытия и отклонения от обычного времени разгрузки), а также шаг моделирования.

Визуализация моделируемого процесса должна предусматривать показ очередей у разгрузочных кранов, приход судов в порт и их отход после разгрузки. Должен быть показан также список произведенных разгрузок, в котором указывается название разгруженного судна, время его прихода в порт и время ожидания в очереди на разгрузку, время начала разгрузки и ее продолжительность.

По окончании моделирования должна быть выведена итоговая статистика: число разгруженных судов, средняя длина очереди на разгрузку, среднее время ожидания в очереди, максимальная и средняя задержка разгрузки, общая сумма выплаченного штрафа. Результат сбора статистики должен быть выведен в текстовый файл.

# **ПЕРВЫЙ РАЗДЕЛ**

## **Формулировка задания**

Вариант 22

Разработать программу для обеспечения продажи туров.

## **Теоретический аспект задач**

Перед реализацией программы необходимо теоретически определить структуру. Для этого принято решение создать абстрактный класс «Tour», который будет представлять общую модель тура. Этот класс будет содержать свойства, определяющие тур как сущность: его идентификационный номер, название тура, дата начала, продолжительность тура и его цена. Кроме того, класс «Tour» будет реализовывать метод «Print», который будет выводить информацию о туре в презентабельном виде, а также реализует интерфейс «IPrintable».

Далее будут созданы три наследующих класса от «Tour», каждый из которых будет представлять отдельный тип тура (пляжный тур, городской тур, приключенческий тур). Эти классы будут содержать дополнительную информацию о типе тура, а метод «Print» будет переопределен для каждого из них. В каждом из этих классов будет добавлено статическое поле, представляющее цену тура для конкретного типа, и это поле будет иметь уникальное значение для каждого типа тура.

## **Формализация задачи**

Далее представлена диаграмма классов (Рисунок 2).

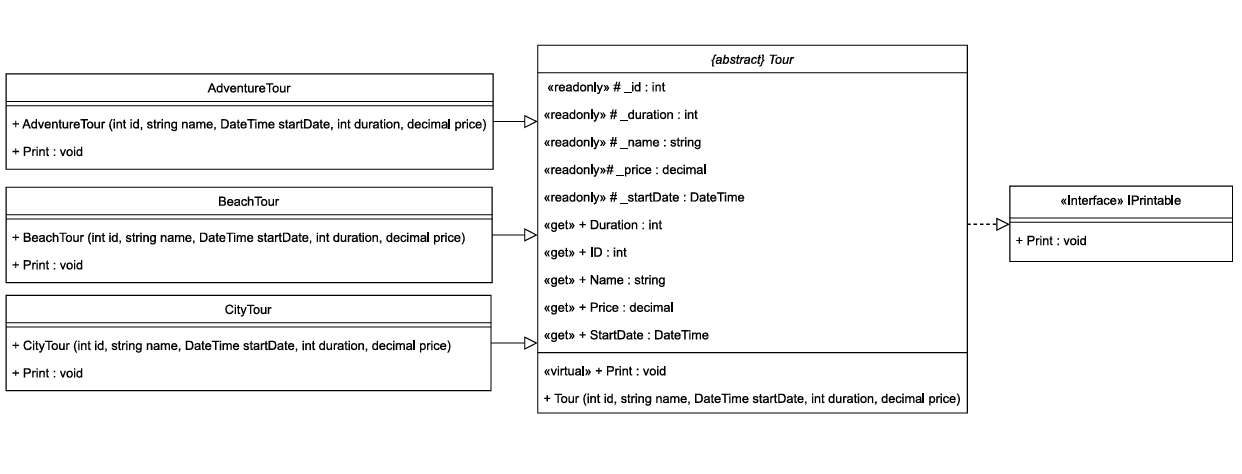


Рисунок 2 – Диаграмма классов для первого раздела

Основой программы является абстрактный класс «Tour», который описывает общую модель тура. Поля этого класса включают в себя основные свойства тура (идентификационный номер, название, дата начала, продолжительность и цена). Конструктор этого класса инициализирует эти поля значениями, и они могут быть доступны только для чтения.

Кроме того, в классе «Tour» есть метод «Print», который реализует интерфейс «IPrintable» и выводит информацию о туре в презентабельном виде. Также реализована логика для определения типа тура через методы классов-наследников.

Для конкретных типов туров (пляжный тур, городской тур, приключенческий тур) создаются производные классы «BeachTour», «CityTour» и «AdventureTour». Эти классы содержат статическое поле «Price», которое задает стоимость тура для каждого типа. Конструктор каждого из этих классов передает параметры в конструктор базового класса, поскольку все эти параметры одинаковы для всех типов туров. Метод «Print» в каждом из классов перегружается, чтобы дополнительно выводить цену тура, после вызова метода «Print» базового класса.

Программа выводит информацию о турах, включая их цену, и позволяет пользователю легко различать типы туров, которые предлагаются для продажи.

## ****Спецификация классов****

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_id | int | protected | Идентификационный номер тура |
| \_name | string | protected | Название тура |
| \_startDate | DateTime | protected | Дата начала тура |
| \_duration | int | protected | Продолжительность тура (в днях) |
| \_price | decimal | protected | Цена тура |

Таблица 1 – Описание полей класса Tour

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| Tour | - | public | int id, string name, DateTime startDate, int duration, decimal price | - | Конструктор класса для инициализации полей |
| Print | void | public | - | - | Вывод информации о туре |

Таблица 2 – Описание методов класса Tour

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_price | decimal | private | Цена пляжного тура |

Таблица 3 – Описание полей класса BeachTour

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| BeachTour | - | public | int id, string name, DateTime startDate, int duration, decimal price | - | Конструктор для инициализации полей класса |
| print | void | public | - | - | Переопределение метода для вывода информации о пляжном туре |

Таблица 4 – Описание методов класса BeachTour

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_price | decimal | private | Цена городского тура |

Таблица 5 – Описание полей класса CityTour

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| CityTour | - | public | int id, string name, DateTime startDate, int duration, decimal price | - | Конструктор для инициализации полей класса |
| Print | void | public | - | - | Переопределение метода для вывода информации о городском туре |

Таблица 6 – Описание методов класса CityTour

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_price | decimal | private | Цена приключенческого тура |

Таблица 7 – Описание полей класса AdventureTour

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| AdventureTour | - | public | int id, string name, DateTime startDate, int duration, decimal price | - | Конструктор для инициализации полей класса |
| Print | void | public | - | - | Переопределение метода для вывода информации о приключенческом туре |

Таблица 8 – Описание методов класса AdventureTour

## **Руководство оператора**

Программа для пользователя является демонстрационной. Все данные, выводящиеся на экран, задаются программистом, пользователю остается лишь запустить программу и увидеть результаты работы программы на экране.

## **Руководство программиста**

Все логически отдельные структуры находятся в раздельных файлах. Название файлов интерфейсов начинается с буквы I. Для получения данных о полях экземпляров классов извне следует использовать реализованные геттеры. Также поля используют стереотип «только для чтения», поэтому присваивание значений новому экземпляру извне происходит лишь однажды при инициализации. Класс Tour является абстрактным, поэтому создание экземпляра класса Tour является недопустимым.

## Контрольный пример

Далее представлены результаты работы программы (Рисунок 3).

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 3 – Контрольный пример работы программы

Пояснение: в методе «Main» создаются три различных тура («BeachTour», «CityTour», «AdventureTour»), и через конструктора им присваиваются свои данные. Затем для каждого объекта по очереди вызывается метод Print, который выводит на экран пользователя информацию о созданных турах. Информация включает идентификационный номер тура, название, дату начала, продолжительность тура, цену и тип тура (например, «Beach Tour», «City Tour», «Adventure Tour»).

## Листинг программы

Program.cs

namespace TourSales

{

internal class Program

{

static void Main(string[] args)

{

BeachTour beachTour = new BeachTour(1, "Maldives Getaway", new DateTime(2024, 12, 1), 7, 49999.99M);

CityTour cityTour = new CityTour(2, "Paris Explorer", new DateTime(2024, 5, 15), 5, 34999.99M);

AdventureTour adventureTour = new AdventureTour(3, "Amazon Jungle Trek", new DateTime(2024, 8, 20), 10, 59999.99M);

beachTour.Print();

Console.WriteLine();

cityTour.Print();

Console.WriteLine();

adventureTour.Print();

}

}

}

IPrintable.cs

namespace TourSales

{

internal interface IPrintable

{

void Print();

}

}

Tour.cs

namespace TourSales

{

internal abstract class Tour : IPrintable

{

protected readonly int \_id; // ID тура

protected readonly string \_name; // Название тура

protected readonly DateTime \_startDate; // Дата начала тура

protected readonly int \_duration; // Продолжительность тура в днях

protected readonly decimal \_price; // Цена тура

public int ID { get { return \_id; } }

public string Name { get { return \_name; } }

public DateTime StartDate { get { return \_startDate; } }

public int Duration { get { return \_duration; } }

public decimal Price { get { return \_price; } }

public Tour(int id, string name, DateTime startDate, int duration, decimal price)

{

\_id = id;

\_name = name;

\_startDate = startDate;

\_duration = duration;

\_price = price;

}

public virtual void Print()

{

Console.WriteLine("Tour ID: {0}\nName: {1}\nStart Date: {2}\nDuration: {3} days\nPrice: {4} Р",

ID, Name, StartDate.ToShortDateString(), Duration, Price);

}

}

}

AdventureTour.cs

namespace TourSales

{

internal class AdventureTour : Tour

{

public AdventureTour(int id, string name, DateTime startDate, int duration, decimal price)

: base(id, name, startDate, duration, price) { }

public override void Print()

{

base.Print();

Console.WriteLine("Type: Adventure Tour");

}

}

}

CityTour.cs

namespace TourSales

{

internal class CityTour : Tour

{

public CityTour(int id, string name, DateTime startDate, int duration, decimal price)

: base(id, name, startDate, duration, price) { }

public override void Print()

{

base.Print();

Console.WriteLine("Type: City Tour");

}

}

}

BeachTour.cs

namespace TourSales

{

internal class BeachTour : Tour

{

public BeachTour(int id, string name, DateTime startDate, int duration, decimal price)

: base(id, name, startDate, duration, price) { }

public override void Print()

{

base.Print();

Console.WriteLine("Type: Beach Tour");

}

}

}

# ВТОРОЙ РАЗДЕЛ

## **Формулировка задания**

Вариант Г-44-3

Для заданного двудольного графа найти число совершенных паросочетаний P и одно из наибольших паросочетаний. Задачу решить в общем виде. В качестве контрольного примера использовать задание вашего варианта.

Изображение выглядит как линия, искусство

Автоматически созданное описание

Рисунок 4 – Задание в качестве контрольного варианта

## **Теоретический аспект задачи**

Для решения поставленной задачи требуется изучить принципы нахождения всех паросочетаний в двудольном графе, а также определить число совершенных паросочетаний и одно из наибольших паросочетаний, используя методы теории графов.

Двудольный граф — это граф, в котором множество вершин делится на две непересекающиеся части, так что все рёбра соединяют вершины из разных частей. Для решения задачи необходимо рассматривать все возможные рёбра между вершинами двух частей и искать максимальные паросочетания, то есть такие подмножества рёбер, где каждому элементу из одной части графа соответствует ровно одно ребро из другой части. Важно, что совершенные паросочетания существуют только в графах с четным числом вершин, поскольку для этого требуется, чтобы все вершины были задействованы в рёбрах.

Для нахождения числа совершенных паросочетаний и вывода одного из наибольших паросочетаний, применяется метод поиска всех паросочетаний в графе с последующей сортировкой и фильтрацией по размеру. В этом контексте также учитываются дополнительные параметры, такие как сортировка по количеству пар в паросочетаниях и нахождение максимальных по размеру паросочетаний.

1. Таким образом, решение задачи включает в себя реализацию методов для:
2. Ввода вершин и рёбер;
3. Поиска всех возможных паросочетаний;
4. Сортировки паросочетаний по размеру и выборки одного из наибольших;
5. Вычисления числа совершенных паросочетаний в графе.

Эти условия были учтены при разработке, что позволило эффективно решать задачу для различных графов.

## **Формализация задачи**

**Описание классов.**

Далее представлена диаграмма классов (Рисунок 5).

Изображение выглядит как снимок экрана, черный

Автоматически созданное описание

Рисунок 5 – Диаграмма классов для второго раздела

Класс BipartiteGraphs является основной частью реализации задачи и описывает двудольный граф, его вершины и рёбра, а также алгоритмы для поиска паросочетаний и проверки на совершенность паросочетаний. В нем используется представление графа через список смежности, который хранится в поле adjacencyList – это словарь, где ключами являются вершины, а значениями – списки смежных вершин.

Класс предоставляет несколько методов для работы с графом. Метод AddVertex добавляет новую вершину в граф, при этом, если вершина уже существует, она не добавляется повторно. Метод RemoveVertex удаляет вершину и все рёбра, связанные с этой вершиной, из списка смежности других вершин. Метод AddEdge добавляет рёбра между двумя существующими вершинами, а метод RemoveEdge удаляет ребро между двумя вершинами, если оно существует. Для ввода данных, метод InputAllVertex позволяет пользователю вводить вершины, пока не будет введена команда "END", а метод InputAllEdge позволяет вводить рёбра между уже добавленными вершинами до команды "END". Метод PrintAdjacencyList выводит список смежности графа, показывая для каждой вершины все вершины, с которыми она соединена рёбрами.

Основные алгоритмы нахождения паросочетаний реализованы через методы FindMatchings, FindAllMatchings и другие. Метод FindMatchings рекурсивно находит все возможные паросочетания в графе, перебирая рёбра и добавляя их в текущее паросочетание, если обе вершины еще не использованы. После добавления ребра метод продолжает искать остальные паросочетания, пока не переберет все возможные рёбра. Каждый найденный набор рёбер, представляющий паросочетание, добавляется в список allMatchings. Метод FindAllMatchings собирает все рёбра графа и передает их в FindMatchings для поиска всех возможных паросочетаний. Метод PrintAllMatchings выводит все найденные паросочетания в формате списков рёбер (vertex1, vertex2). Метод PrintMaxMatching находит и выводит одно из наибольших паросочетаний (по числу рёбер), сортируя все паросочетания по количеству рёбер и выводя одно из самых больших. Метод IsPerfectMatching проверяет, является ли граф совершенным паросочетанием, то есть паросочетанием, в котором количество рёбер равно половине общего числа вершин в графе. Граф должен иметь чётное количество вершин для существования совершенного паросочетания, и метод ищет такие паросочетания и возвращает их количество или 0, если их нет.

Методы SortMatchingsBySizeAndInside и SortMatchingsBySizeDescending предоставляют функционал сортировки паросочетаний. Первый сортирует паросочетания по количеству рёбер в каждом паросочетании, а внутри каждого паросочетания – по возрастанию рёбер. Второй метод делает аналогичную сортировку, но в порядке убывания числа рёбер. Для вывода отсортированных паросочетаний используются методы PrintSortedMatchingsBySizeAndInside и PrintSortedMatchingsBySizeDescending, которые выводят паросочетания в порядке возрастания и убывания количества рёбер соответственно.

**Описание алгоритмов.**

В классе BipartiteGraphs реализуется решение задачи поиска паросочетаний в двудольном графе. Алгоритм начинается с создания и инициализации графа, в котором вершины и рёбра добавляются с помощью соответствующих методов. Каждый граф представляется в виде списка смежности, где для каждой вершины хранится список её соседей.

После того как граф был создан, алгоритм приступает к поиску паросочетаний. В классе есть рекурсивный метод FindMatchings(), который находит все возможные паросочетания. Он перебирает рёбра графа, добавляя их в текущее паросочетание, если вершины ещё не были использованы. После добавления рёбер в паросочетание, метод рекурсивно продолжает искать другие возможные рёбра, избегая пересечения вершин. Полученные паросочетания сохраняются в список.

Далее, для нахождения наибольших паросочетаний, алгоритм сортирует найденные паросочетания по количеству рёбер. Одно из наибольших паросочетаний выводится с помощью метода PrintMaxMatching().

Для нахождения числа совершенных паросочетаний используется метод IsPerfectMatching(), который проверяет, является ли паросочетание совершённым. Совершённым считается паросочетание, количество рёбер в котором равно половине числа вершин в графе (если число вершин чётное). Метод подсчитывает и выводит количество таких паросочетаний.

Ключевые моменты алгоритма:

1. представление графа: граф хранится как список смежности;
2. поиск всех паросочетаний: рекурсивный перебор рёбер с добавлением в паросочетание;
3. поиск наибольшего паросочетания: сортировка паросочетаний по количеству рёбер и вывод одного из наибольших;
4. число совершенных паросочетаний: поиск совершённых паросочетаний, где количество рёбер равно половине числа вершин.

Программа эффективно решает задачу нахождения всех паросочетаний, одно из наибольших и вычисления числа совершенных паросочетаний для заданного двудольного графа.

## Спецификация классов

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| adjacencyList | Dictionary<string, List<string>> | private | Список смежности, хранящий вершины и рёбра графа |

Таблица 9 – Описание полей класса BipartiteGraphs

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| BipartiteGraphs | - | public | - | - | Конструктор класса, инициализирует список смежности |
| AddVertex | void | public | string vertex | - | Добавляет вершину в граф, если она ещё не существует |
| RemoveVertex | void | public | string vertex | - | Удаляет вершину из графа, а также все рёбра, связанные с ней |
| AddEdge | void | public | string vertex1, string vertex2 | - | Добавляет ребро между двумя вершинами, если они существуют |
| RemoveEdge | void | public | string vertex1, string vertex2 | - | Удаляет ребро между двумя вершинами, если оно существует |
| InputAllVertex | void | public | - | - | Вводит все вершины для графа, завершение ввода по команде "END" |
| InputAllEdge | void | public | - | - | Вводит рёбра графа в формате пар вершин, завершение ввода по команде "END" |
| PrintAdjacencyList | void | public | - | - | Выводит список смежности для всех вершин графа |
| FindMatchings | void | private | List<Tuple<string, string>> currentMatching, List<List<Tuple<string, string>>> allMatchings, HashSet<string> usedVertices, List<Tuple<string, string>> allEdges | - | Рекурсивный метод для нахождения всех паросочетаний в графе |
| FindAllMatchings | List<List<Tuple<string, string>>> | public | - | - | Находит все возможные паросочетания в графе |
| PrintAllMatchings | void | public | List<List<Tuple<string, string>>> allMatchings | - | Выводит все найденные паросочетания в графе |
| FindAndPrintAllMatchings | void | public | - | - | Находит и выводит все паросочетания в графе |
| SortMatchingsBySizeAndInside | List<List<Tuple<string, string>>> | public | List<List<Tuple<string, string>>> allMatchings | - | Сортирует паросочетания по количеству пар и сортирует пары внутри каждого паросочетания |
| PrintSortedMatchingsBySizeAndInside | void | public | - | - | Выводит отсортированные по размеру и внутреннему порядку паросочетания |
| PrintMaxMatching | void | public | - | - | Выводит одно из наибольших паросочетаний (по количеству пар) |
| SortMatchingsBySizeDescending | List<List<Tuple<string, string>>> | public | List<List<Tuple<string, string>>> allMatchings | - | Сортирует паросочетания по убыванию числа пар и внутри паросочетаний |
| PrintSortedMatchingsBySizeDescending | void | public | - | - | Выводит отсортированные паросочетания по убыванию числа пар |

Таблица 10 – Описание методов класса BipartiteGraphs

## **Руководство оператора**

После запуска программа предложит пользователю ввести вершины и рёбра графа.

1. Ввод вершин: вершины вводятся по одной, каждая на новой строке, через клавишу «Enter». Для завершения ввода вершин необходимо ввести команду «END»;
2. Ввод рёбер: далее программа предложит ввести рёбра. Для этого нужно указать две вершины, которые соединяет данное ребро, разделяя их пробелом;
   1. Если одна из вершин не существует в графе, программа предложит добавить её. Для этого нужно подтвердить добавление, введя “Y”, или отказаться, введя “N”;
   2. Если пользователь согласится добавить вершину, она будет добавлена в граф, и программа попросит повторно ввести ребро с уже добавленной вершиной;
3. Автоматическое построение графа: после завершения ввода вершин и рёбер, программа автоматически построит граф и выведет на экран список смежности, представляющий введённую структуру графа;
4. Поиск паросочетаний: программа также выполнит поиск всех паросочетаний в графе, выведет одно из максимальных паросочетаний и рассчитает количество совершенных паросочетаний, если таковые имеются.

Таким образом, процесс взаимодействия с программой включает в себя последовательный ввод данных, подтверждения на добавление вершин и рёбер, и автоматическую обработку для построения и анализа графа.

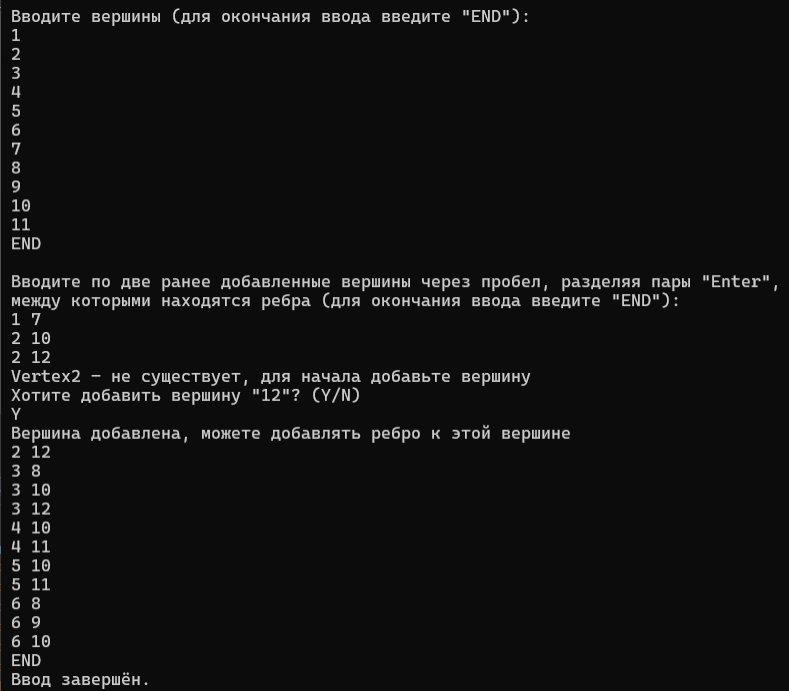


Рисунок 4.1 – Интерфейс ввода вершин и ребер

Изображение выглядит как текст, снимок экрана, Шрифт, черный

Автоматически созданное описание

Рисунок 4.2 – Вывод графа



Рисунок 4.3 – Итоговый вывод

На предоставленных рисунках видно, что все действия, требуемые от пользователя, подробно описываются в консоли. Пользователю необходимо поочерёдно ввести вершины графа, а затем рёбра, соединяющие эти вершины. Если при вводе ребра будет указана вершина, которая отсутствует в графе, программа автоматически уведомит об этом пользователя и предложит добавить новую вершину, предоставляя возможность подтвердить или отказаться от этого действия.

## **Руководство программиста**

## Введение

Проект «BipartiteGraphs» представляет собой консольное приложение на языке C#, предназначенное для работы с двудольными графами. Программа позволяет пользователю вводить вершины и рёбра, строить граф, выводить список смежности, находить паросочетания (включая наибольшее) и определять число совершенных паросочетаний.

## Структура проекта

* Class1.cs: Основной функционал для работы с графами, включая создание, редактирование и анализ графов;
* Program.cs: Главная программа, управляющая взаимодействием с пользователем.

## Основные классы

Класс BipartiteGraphs

Этот класс предоставляет функционал для работы с двудольными графами.

Конструкторы:

* BipartiteGraphs() – инициализирует объект, создавая пустой граф (в виде словаря смежности).

Основные методы:

* 1. Работа с вершинами
* AddVertex(string vertex)

Добавляет вершину в граф. Если вершина уже существует, ничего не происходит.

* RemoveVertex(string vertex)

Удаляет вершину и все рёбра, связанные с ней.

* 1. Работа с рёбрами
* AddEdge(string vertex1, string vertex2)

Добавляет ребро между двумя вершинами. Если хотя бы одна из вершин не существует, пользователю предлагается её создать.

* RemoveEdge(string vertex1, string vertex2)

Удаляет ребро между двумя указанными вершинами, если оно существует.

* 1. Ввод данных
* InputAllVertex()

Позволяет пользователю вводить вершины графа. Для завершения ввода используется команда END.

* InputAllEdge()

Позволяет пользователю вводить рёбра графа. Для завершения ввода используется команда END.

* 1. Вывод данных
* PrintAdjacencyList()

Выводит список смежности графа.

* 1. Работа с паросочетаниями
* FindAllMatchings()

Находит все паросочетания графа.

* PrintAllMatchings(List<List<Tuple<string, string>>> allMatchings)

Выводит найденные паросочетания.

* FindAndPrintAllMatchings()

Находит и выводит все паросочетания.

* SortMatchingsBySizeAndInside(List<List<Tuple<string, string>>> allMatchings)

Сортирует паросочетания по количеству пар и внутреннему порядку.

* PrintSortedMatchingsBySizeAndInside()

Выводит отсортированные паросочетания.

* PrintMaxMatching()

Выводит одно из максимальных паросочетаний.

* IsPerfectMatching()

Определяет число совершенных паросочетаний в графе. Если граф не подходит для поиска совершенных паросочетаний, метод уведомляет пользователя.

## Класс Program

Главный класс программы. Реализует взаимодействие с пользователем.

Точка входа:

static void Main(string[] args)

Алгоритм работы программы:

1. Инициализация графа: BipartiteGraphs graphs = new BipartiteGraphs();
2. Ввод вершин: graphs.InputAllVertex();
3. Ввод рёбер: graphs.InputAllEdge();
4. Вывод списка смежности графа: graphs.PrintAdjacencyList();
5. Вывод максимального паросочетания: graphs.PrintMaxMatching();
6. Вывод числа совершенных паросочетаний: graphs.IsPerfectMatching().

## Особенности и ограничения

1. Обработка ошибок:

Программа контролирует ввод и уведомляет о неверных данных (например, если вершина или ребро не существует).

1. Двудольные графы:

Методы ориентированы на работу с двудольными графами, но программа может работать с произвольными графами в случае небольшой доработки.

1. Сложность:

Алгоритмы нахождения паросочетаний могут быть ресурсозатратными для больших графов.

## Рекомендации

* Проверяйте корректность вводимых данных (например, избегайте дублирования рёбер).
* Используйте уникальные имена для вершин.
* Для работы с крупными графами может потребоваться оптимизация алгоритмов.

## Контрольный приме

Далее представлены результаты работы программы (Рисунки 5.1–5.3).

Изображение выглядит как линия, треугольник, оригами

Автоматически созданное описание

Рисунок 5.1 - Контрольный пример для второго раздела

Исходные данные:

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

Рисунок 5.2– Вводимые данные для контрольного примера

Изображение выглядит как текст, Шрифт, снимок экрана, типография

Автоматически созданное описание

Рисунок 5.3 – Результат работы программы для контрольного примера

Пояснение: была запущена программа с начальными данными, где вершины именовались от 1 до 12 начиная с левого столбца –> вниз левого столбца, верха правого столбца в его низ.

## Листинг кода

**Program.cs**

namespace BipartiteGraphs;

class Program

{

static void Main(string[] args)

{

BipartiteGraphs graphs = new BipartiteGraphs();

graphs.InputAllVertex();

graphs.InputAllEdge();

Console.WriteLine("Ваш введенный граф:\n");

graphs.PrintAdjacencyList();

//graphs.PrintSortedMatchingsBySizeDescending();

graphs.PrintMaxMatching();

Console.WriteLine($"Число совершенных паросочетаний: {graphs.IsPerfectMatching()}");

}

}

**FunctionBipartitegrapg.cs**

namespace BipartiteGraphs

{

internal class BipartiteGraphs

{

private Dictionary<string, List<string>> adjacencyList;

public BipartiteGraphs()

{

adjacencyList = new Dictionary<string, List<string>>();

}

public void AddVertex(string vertex)

{

if (!adjacencyList.ContainsKey(vertex))

{

adjacencyList[vertex] = new List<string>();

}

}

public void RemoveVertex(string vertex)

{

if (adjacencyList.ContainsKey(vertex))

{

foreach (var key in adjacencyList.Keys.ToList())

{

adjacencyList[key].Remove(vertex);

}

adjacencyList.Remove(vertex);

}

}

public void AddEdge(string vertex1, string vertex2)

{

if (!adjacencyList.ContainsKey(vertex1))

{

Console.WriteLine("Vertex1 - не существует, для начала добавьте вершину");

Console.WriteLine($"Хотите добавить вершину \"{vertex1}\"? (Y/N)");

string input;

do

{

input = Console.ReadLine();

if (input == "Y")

{

this.AddVertex(vertex1);

Console.WriteLine("Вершина добавлена, можете добавлять ребро к этой вершине");

return;

}

else if (input == "N") return;

else Console.WriteLine("Введите \"Y\" или \"N\"");

} while (true);

return;

}

if (!adjacencyList.ContainsKey(vertex2))

{

Console.WriteLine("Vertex2 - не существует, для начала добавьте вершину");

Console.WriteLine($"Хотите добавить вершину \"{vertex2}\"? (Y/N)");

string input;

do

{

input = Console.ReadLine();

if (input == "Y")

{

this.AddVertex(vertex2);

Console.WriteLine("Вершина добавлена, можете добавлять ребро к этой вершине");

return;

}

else if (input == "N") return;

else Console.WriteLine("Введите \"Y\" или \"N\"");

} while (true);

return;

}

if (!adjacencyList[vertex1].Contains(vertex2)){

adjacencyList[vertex1].Add(vertex2);

adjacencyList[vertex2].Add(vertex1);

}

}

public void RemoveEdge(string vertex1, string vertex2)

{

if (!adjacencyList.ContainsKey(vertex1))

{

Console.WriteLine($"Вершина {vertex1} не существует.");

return;

}

if (!adjacencyList.ContainsKey(vertex2))

{

Console.WriteLine($"Вершина {vertex2} не существует.");

return;

}

if (adjacencyList[vertex1].Contains(vertex2))

{

adjacencyList[vertex1].Remove(vertex2); // Удаляем ребро из vertex1

Console.WriteLine($"Ребро между {vertex1} и {vertex2} удалено.");

}

else

{

Console.WriteLine($"Ребро между {vertex1} и {vertex2} не существует.");

}

if (adjacencyList[vertex2].Contains(vertex1))

{

adjacencyList[vertex2].Remove(vertex1); // Удаляем ребро из vertex2

}

else

{

Console.WriteLine($"Ребро между {vertex2} и {vertex1} не существует.");

}

}

public void InputAllVertex()

{

Console.WriteLine("Вводите вершины (для окончания ввода введите \"END\"):");

string vertex;

do

{

vertex = Console.ReadLine();

if (vertex != "END") this.AddVertex(vertex);

else

{

Console.WriteLine(' ');

break;

}

} while (true);

}

public void InputAllEdge()

{

Console.WriteLine("Вводите по две ранее добавленные вершины через пробел, разделяя пары \"Enter\",\nмежду которыми находятся ребра (для окончания ввода введите \"END\"):");

string input;

do

{

input = Console.ReadLine();

if (input == "END")

{

Console.WriteLine("Ввод завершён.\n");

break;

}

try

{

// Разбиваем ввод на части

var parts = input.Split(' ');

if (parts.Length != 2)

{

throw new FormatException("Нужно ввести ровно две вершины через пробел.");

}

string vertex1 = parts[0];

string vertex2 = parts[1];

// Добавляем ребро

this.AddEdge(vertex1, vertex2);

}

catch (FormatException ex)

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

catch (Exception ex)

{

Console.WriteLine($"Неожиданная ошибка: {ex.Message}");

}

} while (true);

}

public void PrintAdjacencyList()

{

foreach (var vertex in adjacencyList)

{

Console.WriteLine($"{vertex.Key}: {string.Join(", ", vertex.Value)}");

Console.WriteLine();

}

}

private void FindMatchings(List<Tuple<string, string>> currentMatching, List<List<Tuple<string, string>>> allMatchings, HashSet<string> usedVertices, List<Tuple<string, string>> allEdges)

{

// Если текущее паросочетание не пусто, добавляем его в список

if (currentMatching.Count > 0)

{

// Чтобы избежать дублирования, проверяем, если такое паросочетание уже существует

var sortedMatching = currentMatching.OrderBy(m => m.Item1).ThenBy(m => m.Item2).ToList();

if (!allMatchings.Any(matching => matching.SequenceEqual(sortedMatching)))

{

allMatchings.Add(sortedMatching);

}

}

// Перебираем все рёбра

foreach (var edge in allEdges)

{

var vertex1 = edge.Item1;

var vertex2 = edge.Item2;

// Проверяем, были ли уже использованы эти вершины

if (!usedVertices.Contains(vertex1) && !usedVertices.Contains(vertex2))

{

// Добавляем текущее ребро в паросочетание

currentMatching.Add(edge);

usedVertices.Add(vertex1);

usedVertices.Add(vertex2);

// Рекурсивно ищем другие паросочетания

FindMatchings(currentMatching, allMatchings, usedVertices, allEdges);

// Возвращаемся в предыдущее состояние

currentMatching.RemoveAt(currentMatching.Count - 1);

usedVertices.Remove(vertex1);

usedVertices.Remove(vertex2);

}

}

}

public List<List<Tuple<string, string>>> FindAllMatchings()

{

List<List<Tuple<string, string>>> allMatchings = new List<List<Tuple<string, string>>>();

List<Tuple<string, string>> allEdges = new List<Tuple<string, string>>();

// Собираем все рёбра

foreach (var vertex in adjacencyList.Keys)

{

foreach (var neighbor in adjacencyList[vertex])

{

if (vertex.CompareTo(neighbor) < 0) // Чтобы избежать симметричных рёбер

{

allEdges.Add(Tuple.Create(vertex, neighbor));

}

}

}

FindMatchings(new List<Tuple<string, string>>(), allMatchings, new HashSet<string>(), allEdges);

return allMatchings;

}

// Метод для вывода всех паросочетаний

public void PrintAllMatchings(List<List<Tuple<string, string>>> allMatchings)

{

if (allMatchings.Count == 0)

{

Console.WriteLine("Паросочетаний в данном случае нет, введены неверные данные");

return;

}

Console.WriteLine("Найденные паросочетания:");

foreach (var matching in allMatchings)

{

Console.WriteLine(string.Join(", ", matching.Select(m => $"({m.Item1}, {m.Item2})")));

}

}

// Метод для нахождения и вывода всех паросочетаний

public void FindAndPrintAllMatchings()

{

List<List<Tuple<string, string>>> allMatchings = FindAllMatchings(); // Находим все паросочетания

PrintAllMatchings(allMatchings); // Выводим все паросочетания

}

// Функция для сортировки паросочетаний по количеству пар

public List<List<Tuple<string, string>>> SortMatchingsBySizeAndInside(List<List<Tuple<string, string>>> allMatchings)

{

// Сортируем паросочетания по количеству пар

var sortedMatchings = allMatchings.OrderBy(matching => matching.Count).ToList();

// Для каждого паросочетания сортируем пары внутри по возрастанию

foreach (var matching in sortedMatchings)

{

matching.Sort((m1, m2) =>

{

int compareFirst = m1.Item1.CompareTo(m2.Item1);

if (compareFirst != 0)

return compareFirst;

return m1.Item2.CompareTo(m2.Item2);

});

}

return sortedMatchings;

}

// Функция для вывода отсортированных паросочетаний по количеству пар и сортировке внутри

public void PrintSortedMatchingsBySizeAndInside()

{

List<List<Tuple<string, string>>> allMatchings = FindAllMatchings(); // Находим все паросочетания

// Сортируем паросочетания по количеству пар и внутри каждого паросочетания

List<List<Tuple<string, string>>> sortedMatchings = SortMatchingsBySizeAndInside(allMatchings);

// Выводим отсортированные паросочетания

PrintAllMatchings(sortedMatchings);

}

//Метод для вывода одного из больших паросочетаний

public void PrintMaxMatching()

{

List<List<Tuple<string, string>>> allMatchings = FindAllMatchings(); // Находим все паросочетания

// Сортируем паросочетания по количеству пар и внутри каждого паросочетания

List<List<Tuple<string, string>>> sortedMatchings = SortMatchingsBySizeAndInside(allMatchings);

// Проверяем, есть ли хотя бы одно паросочетание

if (sortedMatchings.Count > 0)

{

// Выводим только последнюю строку (последнее паросочетание)

var lastMatching = sortedMatchings.Last();

Console.WriteLine($"Одно из наибольших паросочетаний: {lastMatching.Count} пар");

Console.WriteLine(string.Join(", ", lastMatching.Select(m => $"({m.Item1}, {m.Item2})")));

}

else

{

Console.WriteLine("Паросочетаний не найдено.");

}

}

public List<List<Tuple<string, string>>> SortMatchingsBySizeDescending(List<List<Tuple<string, string>>> allMatchings)

{

// Сортируем паросочетания по количеству пар по возрастанию

var sortedMatchings = SortMatchingsBySizeAndInside(allMatchings);

// Инвертируем порядок для сортировки по убыванию

sortedMatchings.Reverse();

return sortedMatchings;

}

public void PrintSortedMatchingsBySizeDescending()

{

List<List<Tuple<string, string>>> allMatchings = FindAllMatchings(); // Находим все паросочетания

// Сортируем паросочетания по убыванию

List<List<Tuple<string, string>>> sortedMatchings = SortMatchingsBySizeDescending(allMatchings);

// Выводим отсортированные паросочетания

PrintAllMatchings(sortedMatchings);

}

//Метод для нахождения числа совершенных паросочетаний

public int IsPerfectMatching()

{

// Находим все паросочетания

List<List<Tuple<string, string>>> allMatchings = FindAllMatchings();

// Если нет паросочетаний, возвращаем 0

if (allMatchings.Count == 0)

{

return 0;

}

// Находим максимальную длину паросочетания

int maxLength = allMatchings.Max(matching => matching.Count);

if (maxLength \* 2 != adjacencyList.Count)

{

Console.WriteLine("Совершенные паросочетания могут быть только у графов с четным количеством вершин");

return 0;

}

// Подсчитываем количество паросочетаний с максимальной длиной

int countMaxLengthMatchings = allMatchings.Count(matching => matching.Count == maxLength);

return countMaxLengthMatchings;

}

}

}

# ТРЕТИЙ РАЗДЕЛ

## **Формулировка задания**

Вариант 23

Имитация карточной игры «Дурак переводной»

Создать программу, имитирующую карточную игру «Дурак переводной». Описать правила игры. Применить их в логике программы.

Создать объектную структуру, подходящую для игры нескольких человек в карты. Пользовательский интерфейс должен обеспечивать ввод ходов и команд. Необходимо сформировать протокол игры с выдачей его по завершении и запоминанием «для истории» (Сбора статистики).

Обеспечить вывод результатов сбора статистики в текстовый файл.

## **Теоретический аспект задачи**

Для решения поставленной задачи требуется изучить правила игры «Дурак», дополнения в варианте игры «Дурак переводной», и применить полученные сведения в логике программы.

«Дурак» — игра для 2-4 человек с колодой из 36 игральных карт старшинством от 6 до туза, где каждому в начале раздается по 6 карт, а нижняя карта из колоды становится козырной, видимой для всех игроков. Все карты, у которых масть совпадает с козырной, также являются козырными. Игру начинает игрок с самым младшим козырем (или самый младший игрок, если ни у кого нет в руке козырей), далее ход идет по часовой стрелке. Ведущий подкидывает своему соседу слева карты из своей руки, при этом он может подкинуть несколько карт, если они одного старшинства, на что оппоненту требуется ответить, покрыв карты из своей руки — покрывать карту можно одной масти и старшинством выше, либо козырем (однако если был подкинут козырь, отвечать можно только козырем большего старшинства, по этой причине козырной туз является в игре небьющейся картой); если же ответить нечем, игрок должен взять карты и пропустить свой ход, если все карты успешно были покрыты, ведущий может продолжать подкидывать карты, которые совпадают по старшинству с уже выложенными, или, если такой возможности нет, оставить подкидывание карт остальным игрокам за столом, но не более 6 карт на покрытие или даже меньше, если у отвечающего меньше 6 карт. Битые карты уходят в сброс — они больше не участвуют в игре, после чего каждому игроку раздаются карты из колоды, пока их не станет по 6 штук в руке (игрокам с большим количеством карт или не участвовавшим в подкидывании карты не раздают). Раздача происходит в следующем порядке: сначала ведущий (чей был ход), затем все остальные игроки, подкидывавшие карты в стол, и затем только отвечающий. Конечная цель игры — полностью избавиться от карт в руке, соответственно, оставшийся с картами в руке проигрывает (остается в «дураках»).

Дополнения «Дурака переводного»: пока отвечающий не начал отбиваться, он может перевести подкинутые карты на следующего по кругу игрока и сделать его отвечающим, если у него есть карты одного старшинства с выложенными в стол, при этом можно выложить как одну, так и несколько карт сразу.

Соответственно, при игре на двоих переводы происходят друг на друга, и круг оппонентов сужен до одного человека. Эти условия и были учтены при создании имитационной модели.

## **Формализация задачи**

**Описание классов.**

Далее представлена диаграмма классов (Рисунок 3.1).

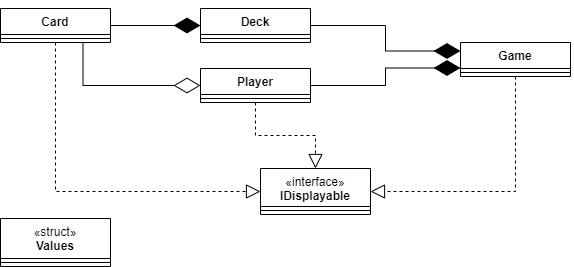


Рисунок 3.1 – Диаграмма классов для второго раздела

На рисунках 3.2 – 3.7 представлены диаграммы отдельных классов.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 3.2 – Диаграмма структуры Values

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 3.3 – Диаграмма интерфейса IDisplayable

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 3.4 – Диаграмма класса Card

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 3.5 – Диаграмма класса Deck

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 3.6 – Диаграмма класса Player

Изображение выглядит как текст, снимок экрана, число, дисплей

Автоматически созданное описание

Рисунок 3.7 – Диаграмма класса Game

Игра является композицией базовых классов. Класс Card является самым базовым и описывает модель игральной карты. Полями этого класса являются свойства игральной карты (старшинство и масть). Card является агрегируемой частью класса Player, описывающего модель игрока, у которого есть одно поле — рука игрока, реализованная как список карт. Также Card явяляется частью композиции класса Deck, описывающей модель колоды карт, полями которого являются список карт — собственно, сама колода карт, и генератор случайных чисел, необходимый для перемешивания колоды. В свою очередь классы Player и Deck являются частями композитного класса Game, который является описанием игровой логики и самой игрой, то есть, весь игровой процесс происходит в этом классе. Полями этого класса являются игровые параметры, необходимые для достаточной реализации игры — колода карт, козырная карта, игроки, кто нападает, кто ходит, игровой стол, сброс, сколько карт было подкинуто, сколько карт было побито, идет ли игра. Классы Card, Player и Game реализуют интерфейс IDisplayable для красивого вывода объекта на экран.

**Описание алгоритмов.**

Что касается обработки различных внутриигровых ситуаций — в классе Game происходит подготовка к игре — перемешивание колоды, раздача карт игрокам, определение козырной карты колоды и игрока, чей ход будет первым по младшему козырю, после чего запускается сам игровой процесс, в котором, путем нажатия требуемых от пользователя клавиш, совершаются действия. Также предусмотрено разделение обработки на действия, которые может совершать нападающий игрок — подкидывать карты из своей руки, передавать хода отбивающемуся игроку и объявлять бито, и действия, которые может совершать отбивающийся игрок — покрывать выложенные карты из своей руки, переводить карты одного старшинства на оппонента (также из своей руки), забирать карты со стола в случае неудачного ответа и передавать ход нападающему игроку в случае удачного ответа.

За каждое из вышеперечисленных действий в игре отвечают отдельные методы класса Game, которые до исполнения самого действия в соответствии с правилами игры «Дурак переводной» проверяют, а позволяет ли текущая игровая ситуация (карты на столе, карты у игроков, карты в колоде) совершить действие. После выполнения действий «объявить бито» и «забрать карты со стола» дополнительно проверяются колода и рукава игроков: если после объявления бито у игроков не осталось ни одной карты, и в колоде тоже не осталось карт, игра считается завершенной вничью; если после взятия карт со стола у нападавшего игрока не осталось ни одной карты, и в колоде тоже не осталось карт, игра считается завершенной с признанием нападавшего игрока победителем.

## Спецификация классов

Описание полей структуры Values представлено в таблице 13.

**Таблица 13**

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| Ranks | Dictionary<byte, string> | public | Перевод числовых значений карт в старшинства |
| Suits | enum | public | Перевод букв, соответствующих мастям, в числа |

Описание полей класса Card представлено в таблице 14.

**Таблица 14**

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_value | byte | private | Старшинство карты в числовом виде |
| \_suit | Values.Suits | private | Масть карты |

Описание методов класса Card представлено в таблице 15.

**Таблица 15**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| Card | - | public | byte val, Values.Suits suit | - | Конструктор класса |
| Display | void | public | - | - | Вывод карты в презентабельном виде |

Описание полей класса Deck представлено в таблице 16.

**Таблица 16**

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_cards | List<Card> | private | Список-колода карт |
| \_rng | Random | private | Генератор случайных чисел для перемешивания колоды |

Описание методов класса Deck представлено в таблице 17.

**Таблица 17**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| Deck | - | public | - | - | Конструктор класса |
| Shuffle | void | private | - | - | Перемешивание колоды карт |
| Size | int | public | - | - | Возврат размера колоды |
| GiveCard | void | public | Player player | - | Отдать карту игроку из колоды |
| RemoveCard | void | private | - | - | Убрать карту из колоды |

Описание полей класса Player представлено в таблице 18.

**Таблица 18**

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_hand | List<Card> | private | Рука игрока |

Описание методов класса Player представлено в таблице 19.

**Таблица 19**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| Player | - | public | - | - | Конструктор класса |
| AddCard | void | public | Card card | - | Добавление карты в руку игрока |
| RemoveCard | void | public | int index | - | Удаление карты из руки игрока по переданному индексу |
| SizeOfHand | int | public | - | - | Подсчет количества карт в руке игрока |
| MinTrump | byte | public | Card trump | - | Определение младшего козыря |
| Display | void | public | - | - | Вывод игрока в презентабельном виде |

Описание полей класса Game представлено в таблице 20.

**Таблица 20**

| Имя | Тип | Модификатор доступа | Назначение |
| --- | --- | --- | --- |
| \_deck | Deck | private | Колода карт |
| \_trump | Card | private | Козырная карта |
| \_p1 | Player | private | Игрок 1 |
| \_p2 | Player | private | Игрок 2 |
| \_attacker | Player | private | Кто нападает в данный момент |
| \_turn | Player | private | Кто ходит в данный момент |
| \_field | (Card, Card)[] | private | Игровой стол |
| \_dropped | List<Card> | private | Сброс |
| \_count | byte | private | Сколько карт было подкинуто |
| \_covered | byte | private | Сколько карт было отбито |
| \_game | bool | private | Идет ли игра |
| \_log | System.IO.StreamWriter | private | Протоколирование игры |

Описание методов класса Game представлено в таблице 21.

**Таблица 21**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| Game | - | public | - | - | Конструктор класса, по совместительству, и сама игра |
| Defender | Player | private | int index | - | Определение отбивающегося игрока в текущем коне |
| Attack | void | private | - | - | Подкинуть карту в стол |
| MoveToDefender | void | private | - | - | Передать ход отбивающемуся игроку |
| Drop | void | private | - | - | Скинуть карты со стола в сброс |
| Cover | void | private | int index | - | Покрыть следующую непокрытую карту на столе |
| Turn | void | private | int index | - | Перевести стол на оппонента |
| Take | void | private | - | - | Взять карты |
| MoveToAttacker | void | private | - | - | Передать ход нападающему игроку |
| Display | void | public | - | - | Вывод игрового стола в презентабельном виде |

## **Руководство оператора**

При запуске пользователя встречает консольный интерфейс, где ему отображается, какие действия он может сделать, в зависимости от его хода, и на какие клавиши ему нужно нажимать. Примеры интерфейса показаны на рисунках 4.1-4.3.

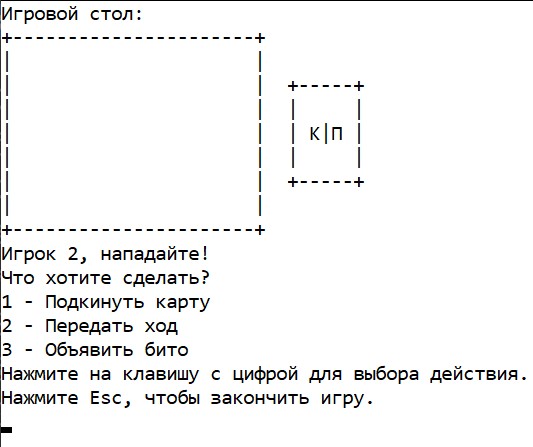


Рисунок 4.1 – Интерфейс нападающего игрока

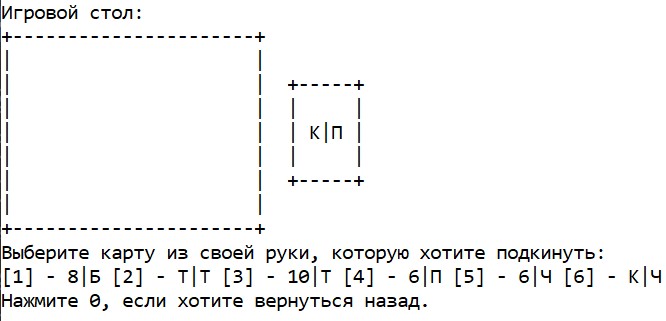


Рисунок 4.2 – Интерфейс выбора карты (доступен при подкидывании, переводе и покрытии)

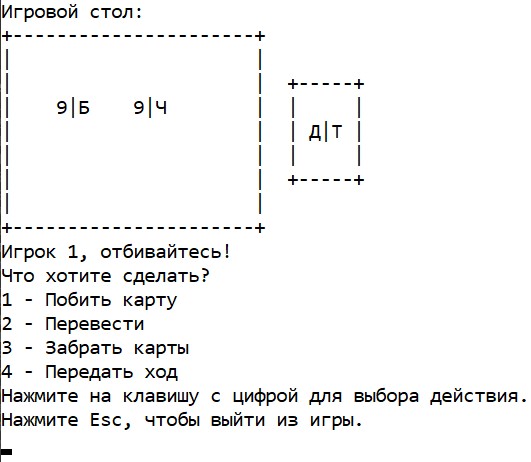


Рисунок 4.3 – Интерфейс отбивающегося игрока

Как можно заметить на рисунках, все действия обозначены клавишей, которую пользователю нужно нажать для совершения желаемого действия, при этом если выбранное действие в данный момент недопустимо, пользователя перебросит обратно на главный интерфейс, где он вновь может выбирать действия.

## **Руководство программиста**

Все логически отдельные структуры находятся в раздельных файлах. Особенности каждого типа данных отображены с помощью комментариев в программе (представлено в листинге). Промежуточная информация о ходах игроков выводится в файл log.txt.

## Контрольный пример

Далее представлены результаты работы программы (Рисунки 5.1 и 5.2).

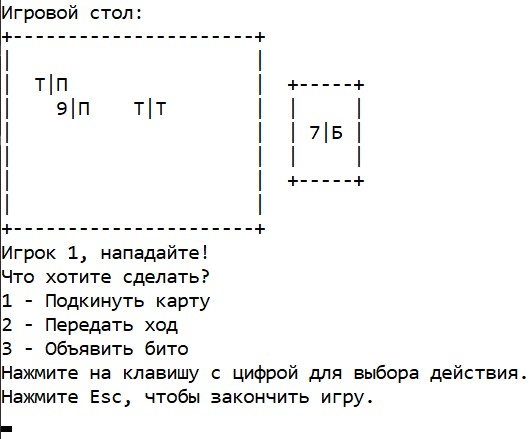


Рисунок 5.1 - Контрольный пример для второго раздела

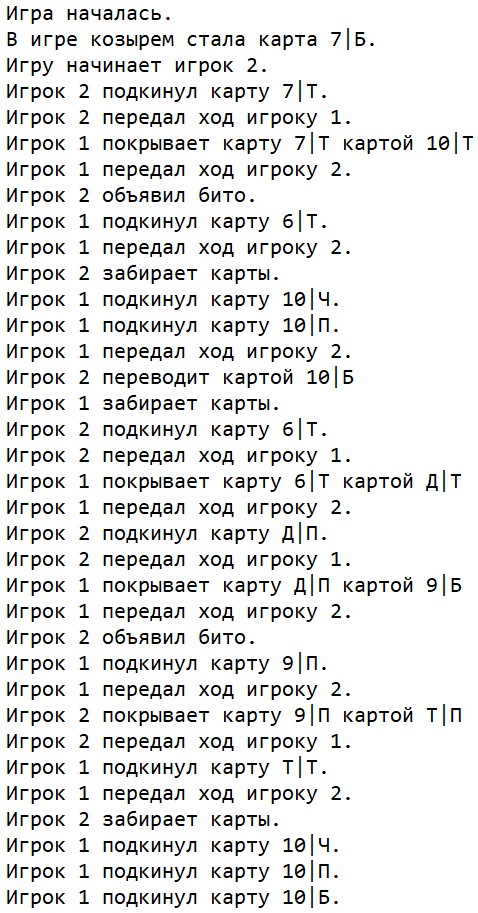


Рисунок 5.2 – протокол для контрольного примера

Пояснение: была запущена демонстративная партия и сыграна от начала до конца, после чего данные игры были записаны в протокол. На рисунке 5.2 можно увидеть, как началась игра, и какие ходы делали игроки.

## Листинг программы

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы были разработаны две программы, основанные на принципах объектно-ориентированного подхода.

Были закреплены навыки работы с файлами, интерфейсами, абстрактными классами, также были закреплены знания о механизмах наследования. Получен опыт в разработке приложений, моделирующих сложные и непостоянные процессы.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Microsoft Learn – сеть разработчиков Microsoft. URL: <https://learn.microsoft.com/ru-ru/dotnet/csharp/> (дата обращения: 20.12.2023)

2. Горячев А. В., Кравчук Д. К., Новакова Н. Е. Объектно-ориентированное моделирование. Учеб. Пособие. СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2010.

3. Новакова Н. Е., Горячев А. В. Моделирование коммуникативных процессов в распределенных САПР. Учеб. Пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2010.

4. Язык UML = The Unified Modeling Language User Guide : руководство пользователя / Г. Буч, Д. Рамбо, А. Джекобсон.

5. Полное руководство по языку программирования С# 12 и платформе .NET 8. URL: <https://metanit.com/sharp/tutorial/> (дата обращения: 09.12.2023)