**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ИБ**

КУРСОВАЯ работа

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

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

|  |  |  |
| --- | --- | --- |
| Студент гр. 3363 |  | Овсейчик Н.И. |
| Преподаватель |  | Новакова Н.Е. |

Санкт-Петербург

2024

1. **закрепление теоретических знаний и практических навыков разработки программ на основе объектно-ориентированного подхода и разработке объектной модели**

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Овсейчик Н.И. | | |
| Группа 3363 | | |
| Тема работы: Разработка приложения на основе принципов объектно-ориентированного подхода | | |
| Содержание пояснительной записки:  Формулировка задания, теоретический аспект задачи, формализация задачи, спецификация программы, руководство оператора, руководство программиста, контрольный пример, листинг кода | | |
| Предполагаемый объем пояснительной записки:  Не менее 25 страниц. | | |
| Дата выдачи задания: 01.10.2024 | | |
| Дата сдачи реферата: 25.11.2024 | | |
| Дата защиты реферата: 25.11.2024 | | |
| Студент |  | Овсейчик Н.И. |
| Преподаватель |  | Новакова Н.Е. |

* 1. **Формулировка задания**

Создать систему классов для описания плоских геометрических фигур: круга, квадрата, прямоугольника. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и поворота на заданный угол.

* 1. **Теоретический аспект задачи**

Задача создания системы классов для описания плоских геометрических фигур (круга, квадрата и прямоугольника) включает в себя несколько важных теоретических аспектов, связанных с основами геометрии, объектно-ориентированным программированием (ООП), а также принципами работы с графикой и геометрическими преобразованиями. Рассмотрим каждый из них подробно:

* + 1. **Геометрические фигуры**

Плоские геометрические фигуры — это объекты, имеющие двумерную форму, которые можно описать с помощью координат в пространстве. В данном случае мы рассматриваем три основных фигуры:

* **Круг** — геометрическая фигура, все точки которой находятся на одинаковом расстоянии от центра. Для описания круга достаточно знать его радиус и координаты центра.
* **Квадрат** — прямоугольник, у которого все стороны равны. Квадрат можно описать с помощью координат одной из его вершин и длины стороны.
* **Прямоугольник** — это четырехугольник, у которого противоположные стороны равны и параллельны. Прямоугольник можно задать координатами двух противоположных вершин (или одной вершины и размеров — ширины и высоты).
  + 1. **Принципы объектно-ориентированного программирования**

Для решения задачи было использовано объектно-ориентированное программирование. В ООП объекты представляют собой экземпляры классов, которые могут обладать как состоянием (в виде полей), так и поведением (в виде методов). В контексте данной задачи:

**Наследование** — позволяет создавать специализированные классы (например, для круга, квадрата и прямоугольника), которые будут наследовать общие свойства от базового класса (например, класс Shape), но при этом иметь свои уникальные характеристики и методы.

**Инкапсуляция** — скрытие реализации конкретных методов и данных за абстракцией. В нашей задаче, например, класс Shape может скрывать детали реализации, такие как расчёт площади или периметра, позволяя пользователю работать только с нужными методами (например, Move, Resize, Rotate).

**Полиморфизм** — позволяет использовать один интерфейс для работы с разными типами объектов. В данном случае, все фигуры (круг, квадрат и прямоугольник) наследуют общий интерфейс от абстрактного класса Shape, что позволяет манипулировать ими одинаковым образом, не заботясь о их типах.

* + 1. **Работа с геометрическими преобразованиями**

Для работы с геометрическими фигурами необходимо учитывать их перемещение, изменение размеров и поворот. Эти операции являются важными аспектами геометрических трансформаций, которые в дальнейшем могут быть использованы для более сложных задач, таких как создание анимаций или моделирование.

* **Перемещение (Translation)** — это изменение координат объекта на плоскости. Для каждой фигуры операция перемещения осуществляется с помощью смещения всех её точек на заданное расстояние по осям X и Y.
* **Изменение размеров (Scaling)** — изменение размера фигуры, при котором её геометрия изменяется пропорционально. Для круга это изменение радиуса, для квадрата и прямоугольника — изменение длины сторон. Важно, что операции масштабирования могут быть одинаковыми по осям или независимыми.
* **Поворот (Rotation)** — операция, изменяющая ориентацию фигуры относительно её центра или другого заданного пункта на плоскости. Поворот выполняется на заданный угол по часовой стрелке или против неё, что важно при изменении ориентации геометрических объектов в 2D-пространстве. Реализация поворота фигуры “Круг” является бессмысленной, так как при повороте на определённый угол “Круг” не изменит свою форму и положение.
  1. **Формализация задачи**

Для описания плоских геометрических фигур были выбраны классы Shape, ShapeManager, Circle, Square, и Rectangle:

* Shape — абстрактный базовый класс, содержащий общие методы и свойства для всех геометрических фигур. От него наследуются конкретные классы.
* ShapeManager — служит для централизованного управления коллекцией фигур. Этот класс управляет списком объектов типа Shape, предоставляя методы для их добавления, удаления, перемещения и отображения.
* Circle — класс, который описывает круг, имеющий радиус и координаты центра.
* Square — класс, описывающий квадрат, для которого важны длина стороны и координаты.
* Rectangle — класс для прямоугольника, который описан двумя сторонами и координатами.
* ShapeInitializer — статический класс, предоставляющий методы для безопасной инициализации фигур (InitializeCircle, InitializeSquare, InitializeRectangle). Он запрашивает необходимые параметры фигуры у пользователя и возвращает соответствующий объект. Это облегчает процесс создания фигур.
* InputHelper — статический класс, предоставляющий вспомогательные методы для ввода данных с консоли. Он обеспечивает безопасный ввод чисел, включая проверку на ошибки и отрицательные значения.

Каждый из этих классов имеет методы для:

1. Shape, Circle, Square, Rectangle:

* Move — перемещение фигуры по плоскости;
* Resize — изменение размера фигуры;
* Rotate — поворот фигуры на заданный угол (не реализован для круга);
* Display — отображение информации о фигуре.

1. ShapeManager:

* AddShape — добавление фигуры в коллекцию;
* RemoveShape — удаление фигуры из коллекции;
* DisplayAll — отображение информации обо всех фигурах;
* MoveAll — перемещение всех фигур в коллекции;
* ResizeAll — изменение размеров всех фигур.

1. ShapeInitializer:

* InitializeCircle — инициализация объекта типа Circle;
* InitializeSquare — инициализация объекта типа Square;
* InitializeRectangle — инициализация объекта типа Rectangle.

1. InputHelper:

* GetDoubleFromUser — безопасный ввод вещественного числа;
* GetPositiveDoubleFromUser — безопасный ввод положительного числа.
  + 1. **Принцип работы с геометрическими фигурами**
* Перемещение (Move): изменяет координаты всех точек фигуры на заданное расстояние.
* Изменение размера (Resize): изменение размеров фигуры с сохранением пропорций.
* Поворот (Rotation): изменение ориентации фигуры относительно центра или заданной точки.

Для упрощения работы с пользовательским вводом и создания объектов фигур в программе добавлены два вспомогательных класса:

* InputHelper обеспечивает безопасный ввод данных (например, чисел или положительных значений) с минимизацией ошибок. Это особенно важно для корректной работы операций перемещения, изменения размеров и поворота.
* ShapeInitializer стандартизирует процесс создания фигур, делая его удобным и структурированным. Он запрашивает параметры фигуры у пользователя, используя методы InputHelper, и возвращает инициализированный объект.
  + 1. **Использование наследования и полиморфизма**

Наследование позволяет определить общие операции для всех типов фигур в базовом классе Shape, а полиморфизм помогает реализовать специфические действия для каждого типа фигуры (например, расчёт площади или периметра). Это упрощает управление различными типами объектов и позволяет добавлять новые фигуры без значительных изменений в основной логике программы.

* + 1. **Диаграмма классов UML**

В этой диаграмме:

* Shape — базовый (абстрактный) класс;
* Circle, Square, Rectangle — классы-наследники;
* ShapeManager — класс, управляющий коллекцией фигур.
* InputHelper и ShapeInitializer — вспомогательные классы для обработки ввода и создания объектов.

Каждый класс имеет свои методы и атрибуты, а также отношения между ними, например, наследование и управление коллекцией объектов (рис.1).

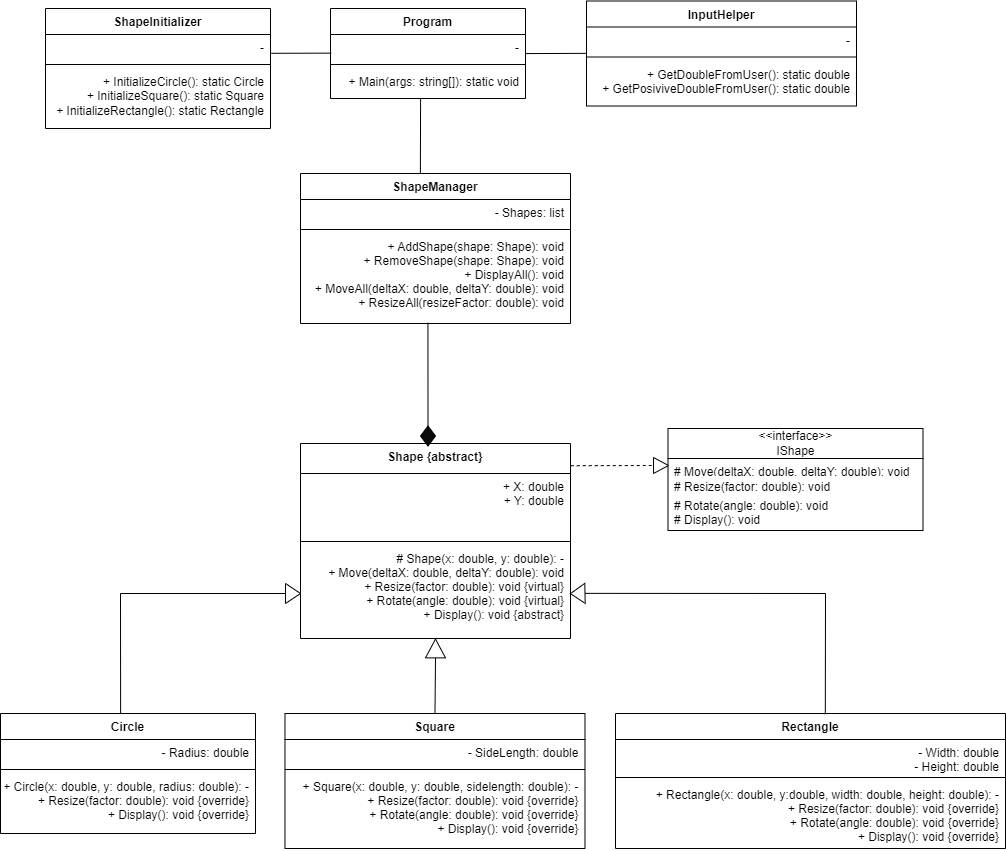


Рисунок 1 — UML диаграмма классов программы

* 1. **Спецификация программы**
     1. **Класс Shape**

Описание методов класса Shape представлено в таблице 1.4.1.1.

Таблица 1.4.1.1 Описание методов класса Shape

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Shape | - | protected | double x,  double y | Конструктор класса. Создает объект фигуры с заданными координатами X и Y. |
| Move | void | public | double deltaX, double deltaY | Перемещение фигуры на заданное расстояние по осям X и Y |
| Resize | virtual void | public | double factor | Изменение размера фигуры на указанный коэффициент |
| Rotate | virtual void | public | double angle | Поворот фигуры на указанный угол |
| Display | abstract void | public | - | Отображение фигуры (вывод информации о ней) |

Описание полей класса Shape представлено в таблице 1.4.1.2.

Таблица 1.4.1.2 Описание полей класса Shape

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| X | double | protected | Координата X фигуры |
| Y | double | protected | Координата Y фигуры |

* + 1. **Класс Circle (наследует Shape)**

Описание методов класса Circle представлено в таблице 1.4.2.1.

Таблица 1.4.2.1 Описание методов класса Circle

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Circle | - | public | double x,  double y,  double radius | Конструктор класса. Создает объект круга с заданными координатами X, Y и радиусом Radius. |
| Display | override void | public | - | Отображение информации о круге |
| Resize | override void | public | double factor | Изменение радиуса круга на заданный коэффициент |

Описание полей класса Circle представлено в таблице 1.4.2.2.

Таблица 1.4.2.2 Описание полей класса Circle

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| Radius | double | private | Радиус круга |

* + 1. **Класс Square (наследует Shape)**

Описание методов класса Square представлено в таблице 1.4.3.1.

Таблица 1.4.3.1 Описание методов класса Square

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Square | - | public | double x,  double y,  double sidelenght | Конструктор класса. Создает объект квадрата с заданными координатами X, Y и длиной стороны SideLength. |
| Display | override void | public | - | Отображение информации о квадрате |
| Resize | override void | public | double factor | Изменение размера квадрата на указанный коэффициент |
| Rotate | override void | public | double angle | Поворот квадрата на заданный угол |

Описание полей класса Square представлено в таблице 1.4.3.2.

Таблица 1.4.3.2 Описание полей класса Square

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| SideLength | double | private | Длина стороны квадрата |

* + 1. **Класс Rectangle (наследует Shape)**

Описание методов класса Rectangle представлено в таблице 1.4.4.1.

Таблица 1.4.4.1 Описание методов класса Rectangle

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Rectangle | - | public | double x,  double y,  double width,  double height | Конструктор класса. Создает объект прямоугольника с заданными координатами X, Y, длиной Width и шириной Height. |
| Display | override void | public | - | Отображение информации о прямоугольнике |
| Resize | override void | public | double factor | Изменение размераа прямоугольника по осям X и Y |
| Rotate | override void | public | double angle | Поворот прямоугольника на заданный угол |

Описание полей класса Rectangle представлено в таблице 1.4.4.2.

Таблица 1.4.4.2 Описание полей класса Rectangle

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| Width | double | private | Ширина прямоугольника |
| Height | double | private | Высота прямоугольника |

* + 1. **Класс ShapeManager**

Описание методов класса ShapeManager представлено в таблице 1.4.5.1.

Таблица 1.4.5.1 Описание методов класса ShapeManager

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| AddShape | void | public | Shape shape | Добавление фигуры в коллекцию |
| RemoveShape | void | public | Shape shape | Удаление фигуры из коллекции |
| DisplayAll | void | public | - | Отображение всех фигур в коллекции. |
| MoveAll | void | public | double deltaX, double deltaY | Перемещение всех фигур в коллекции на заданные расстояния по осям X и Y. |
| ResizeAll | void | public | double resizeFactor | Изменяет размеры всех объектов в коллекции на указанный коэффициент масштабирования. |

Описание полей класса ShapeManager представлено в таблице 1.4.5.2.

Таблица 1.4.5.2 Описание полей класса ShapeManager

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| Shapes | List<Shape> | private | Список всех добавленных фигур (кругов, квадратов, прямоугольников) |

* + 1. **Интерфейс IShape**

Описание методов интерфейса IShape представлено в таблице 1.4.6.1.

Таблица 1.4.6.1 Описание методов интерфейса IShape

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Move | void | protected | double deltaX, double deltaY | Перемещение фигуры на заданное расстояние по осям X и Y |
| Resize | void | protected | double factor | Изменение размера фигуры на указанный коэффициент |
| Rotate | void | protected | double angle | Поворот фигуры на указанный угол. |
| Display | void | protected | - | Отображение фигуры (вывод информации о ней). |

* + 1. **Класс** **ShapeInitializer**

Описание методов класса ShapeInitializer представлено в таблице 1.4.7.1.

Таблица 1.4.7.1 Описание методов интерфейса ShapeInitializer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| InitializeCircle | Circle | public static | - | Инициализация объекта типа Circle, запрашивая координаты и радиус у пользователя. |
| InitializeSquare | Square | public static | - | Инициализация объекта типа Square, запрашивая координаты и длину стороны у пользователя. |
| InitializeRectangle | Rectangle | public static | - | Инициализация объекта типа Rectangle, запрашивая координаты, ширину и высоту у пользователя. |

* + 1. **Класс InputHelper**

Описание методов класса InputHelper представлено в таблице 1.4.8.1.

Таблица 1.4.8.1 Описание методов интерфейса InputHelper

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| GetDoubleFromUser | double | public static | - | Выполнение безопасного ввода числа типа double с консоли, обеспечивая проверку корректности данных. |
| GetPositiveDoubleFromUser | double | public static | - | Выполнение безопасного ввода неотрицательного числа типа double, проверяя корректность ввода. |

* + 1. **Класс** **Program**

Описание методов класса Program представлено в таблице 1.4.9.1.

Таблица 1.4.9.1 Описание методов интерфейса Program

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Main | void | public static | string[] args | Главная точка входа в программу. Выполняет последовательное управление и взаимодействие с классами. |

* 1. **Руководство оператора**

Для написания программы был выбран язык программирования C#, операционная система Windows 11, среда разработки Microsoft Visual Studio.

Программа предназначена для работы с геометрическими фигурами: **кругами**, **квадратами** и **прямоугольниками**. Она позволяет:

* Создавать фигуры;
* Перемещать фигуры по координатной плоскости;
* Изменять размеры фигур;
* Выполнять вращение фигур;
* Просматривать текущие состояния всех фигур.
  + 1. **Запуск программы**

1. Запустите программу в любой совместимой среде разработки, поддерживающей C# (например, Visual Studio);
2. Следуйте инструкциям, выводимым в консоль.
   * 1. **Интерфейс пользователя**

Интерфейс пользователя представляет собой консольный ввод-вывод. После запуска программы пользователь последовательно вводит параметры фигур и управляет их действиями.

* + 1. **Основные этапы работы:**

1. **Инициализация фигур**

* Для создания круга введите координаты центра круга X и Y, затем введите радиус круга (должно быть положительное число);
* Для создания квадрата введите координаты верхнего левого угла квадрата X и Y, затем введите длину стороны квадрата (должно быть положительное число);
* Для создания прямоугольника введите координаты верхнего левого угла прямоугольника X и Y, затем введите ширину и высоту прямоугольника (оба значения должны быть положительными).

1. **Просмотр параметров фигур**

После ввода всех фигур программа выведет их параметры в консоль:

* Тип фигуры;
* Координаты;
* Размеры (радиус, стороны и т.д.).

1. **Перемещение всех фигур**

* Введите смещение по X и Y;
* Фигуры переместятся на указанные расстояния, и их новые координаты отобразятся.

1. **Изменение размеров**

* Введите коэффициент изменения размера (должен быть положительным числом);
* Размеры фигур увеличатся или уменьшатся в указанное количество раз.

1. **Вращение фигур**

Укажите угол поворота для фигур (например, квадрат или прямоугольник).

1. **Финальный просмотр**

После выполнения всех операций программа выведет обновленные параметры всех фигур.

* + 1. **Команды ввода**

Все вводимые данные должны соответствовать ожидаемому формату:

* Координаты и размеры — числа с плавающей запятой;
* Коэффициенты изменения размеров — положительные числа;
* Углы вращения — числа в градусах (положительные или отрицательные).

В случае некорректного ввода программа выведет сообщение об ошибке и предложит повторить ввод.

* 1. **Руководство программиста**

Проект представляет собой библиотеку классов и консольное приложение для работы с геометрическими фигурами: кругами, квадратами и прямоугольниками. Основной функционал включает:

* Создание объектов различных форм;
* Управление фигурами: перемещение, изменение размеров и вращение;
* Отображение информации о фигурах.
  + 1. **Архитектура**

Проект построен с использованием объектно-ориентированного подхода.

* + 1. **Интерфейс IShape**

Определяет базовые операции для всех фигур:

* Move(double deltaX, double deltaY) — перемещение фигуры;
* Resize(double factor) — изменение размера фигуры;
* Rotate(double angle) — вращение фигуры;
* Display() — отображение информации о фигуре.
  + 1. **Абстрактный класс Shape**

Реализует интерфейс IShape и содержит базовые свойства:

* X — координата фигуры по оси X;
* Y — координата фигуры по оси Y.
* Предоставляет базовую реализацию метода Move(). Остальные методы могут быть переопределены в производных классах.
  + 1. **Классы фигур**
       1. **Circle**
* Дополнительное свойство: Radius (радиус круга);
* Реализует методы Resize() и Display().
  + - 1. **Square**
* Свойство: SideLength (длина стороны);
* Реализует методы Resize(), Rotate() и Display().
  + - 1. **Rectangle**
* Свойства: Width (ширина) и Height (высота);
* Реализует методы Resize(), Rotate() и Display().
  + - 1. **Класс ShapeManager**

Управляет коллекцией объектов Shape. Основные методы:

* AddShape(Shape shape) — добавление фигуры;
* RemoveShape(Shape shape) — удаление фигуры;
* DisplayAll() — отображение всех фигур;
* MoveAll(double deltaX, double deltaY) — перемещение всех фигур;
* ResizeAll(double resizeFactor) — изменение размера всех фигур.
  + - 1. **Класс InputHelper**

Упрощает обработку пользовательского ввода. Основные методы:

* GetPositiveDoubleFromUser() — безопасное считывание числа с плавающей запятой;
* GetDoubleFromUser() — считывание положительного числа;
  + - 1. **Класс ShapeInitializer**

Отвечает за создание объектов фигур. Основные методы:

* InitializeCircle() — создание объекта Circle с вводом параметров;
* InitializeSquare() — создание объекта Square;
* InitializeRectangle() — создание объекта Rectangle.

Этот класс отделяет логику инициализации фигур от основной программы, что делает код более структурированным и удобным для расширения.

* + - 1. **Программа (Program)**
* Предоставляет консольный интерфейс для взаимодействия с пользователем;
* Использует классы InputHelper и ShapeInitializer для обработки пользовательского ввода и создания объектов.
* Позволяет инициализировать фигуры, перемещать их, изменять размеры и отображать текущие параметры.
  1. **Пример работы программы**
     1. **Запуск программы**

Пользователь запускает программу, которая предлагает инициализировать геометрические фигуры: Круг, Квадрат и Прямоугольник (Ввод)(рис.2):

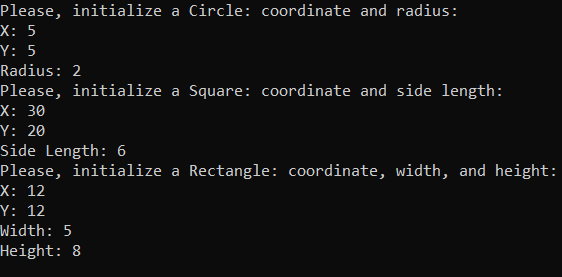


Рисунок 2 — Инициализация фигур

Программа выводит созданные фигуры с их параметрами (Вывод)(рис.3):

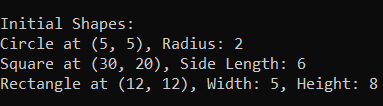


Рисунок 3 — Вывод инициализации фигур

* + 1. **Перемещение фигур**

Далее программа предлагает ввести значения для смещения фигур по осям X и Y (Ввод)(рис.4):

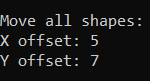


Рисунок 4 — Перемещение фигур по осям X и Y

Программа перемещает фигуры и отображает новые координаты (Вывод)(рис.5):

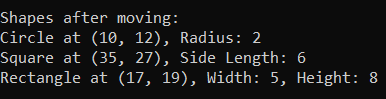


Рисунок 5 — Вывод состояния фигур после перемещения по осям X и Y

* + 1. **Изменение размеров**

Затем программа предложит ввести коэффициент изменения размеров фигур (Ввод)(рис.6):



Рисунок 6 — Изменение размеров фигур

Программа изменяет размеры фигур и выводит обновленные данные(Вывод)(рис.7):

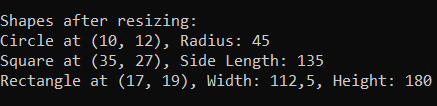


Рисунок 7 — Вывод о состоянии фигур после изменения размеров фигур

* + 1. **Поворот фигур**

После программа предлагает ввести угол поворота для квадрата и прямоугольника и выводит результат поворота (Ввод — Вывод)(рис.8):

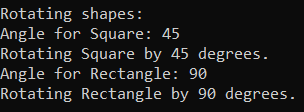


Рисунок 8 — Поворот фигур

* + 1. **Финальный вывод**

Программа отображает финальные данные всех фигур (Вывод)(рис.9):

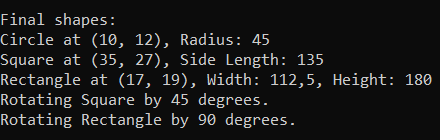


Рисунок 9 — Вывод итогового состояния фигур после преобразований

* 1. **Листинг программы**
     1. **Интерфейс IShape**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// Интерфейс, задающий базовые действия для всех форм.

public interface IShape

{

protected void Move(double deltaX, double deltaY); // Перемещение фигуры

protected void Resize(double factor); // Изменение размера фигура

protected void Rotate(double angle); // Поворот фигуры

protected void Display(); // Отображение данных фигуры

}

* + 1. **Базовый класс Shape**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

// Абстрактный класс, представляющий общие свойства и методы для всех форм

public abstract class Shape : IShape

{

protected double X { get; set; }

protected double Y { get; set; }

// Конструктор базового класса Shape.

protected Shape(double x, double y)

{

X = x;

Y = y;

}

// Перемещает фигуру на заданное смещение.

public void Move(double deltaX, double deltaY)

{

X += deltaX;

Y += deltaY;

}

// Виртуальные методы для изменения размера, поворота и отображения.

// Эти методы переопределяются в дочерних классах.

public virtual void Resize(double factor) { }

public virtual void Rotate(double angle) { }

public abstract void Display();

}

* + 1. **Класс Circle**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

// Класс, представляющий круг.

public class Circle : Shape

{

// Радиус круга

private double Radius { get; set; }

// Конструктор класса Circle.

public Circle(double x, double y, double radius) : base(x, y)

{

Radius = radius;

}

// Изменение размера круга.

public override void Resize(double factor)

{

Radius \*= factor;

if (Radius < 0)

{

Console.WriteLine("ERROR");

}

}

// Вывод информации о круге.

public override void Display()

{

Console.WriteLine($"Circle at ({X}, {Y}), Radius: {Radius}");

}

}

* + 1. **Класс Square**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

// Класс, представляющий квадрат.

public class Square : Shape

{

//Сторона квадрата

private double SideLength { get; set; }

//Конструктор класса Square

public Square(double x, double y, double sidelength) : base(x, y)

{

SideLength = sidelength;

}

//Изменение размера квадрата

public override void Resize(double factor)

{

SideLength \*= factor;

if (SideLength < 0)

{

Console.WriteLine("ERROR");

}

}

//Поворот квадрата

public override void Rotate(double angle)

{

Console.WriteLine($"Rotating Square by {angle} degrees.");

}

//Вывод информации о квадрате

public override void Display()

{

Console.WriteLine($"Square at ({X}, {Y}), Side Length: {SideLength}");

}

}

* + 1. **Класс Rectangle**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

//Класс, представляющий прямоугольник

public class Rectangle : Shape

{

private double Width { get; set; } //Длина прямоугольника

private double Height { get; set; } //Ширина прямоугольника

//Конструктор класса Rectangle

public Rectangle(double x, double y, double width, double height) : base(x, y)

{

Width = width;

Height = height;

}

//Изменение размеров прямоугольника

public override void Resize(double factor)

{

Width \*= factor;

Height \*= factor;

if (Width < 0 || Height < 0)

{

Console.WriteLine("ERROR");

}

}

//Поворот прямоугольника

public override void Rotate(double angle)

{

Console.WriteLine($"Rotating Rectangle by {angle} degrees.");

}

//Вывод информации о прямоугольнике

public override void Display()

{

Console.WriteLine($"Rectangle at ({X}, {Y}), Width: {Width}, Height: {Height}");

}

}

* + 1. **Класс ShapeManager**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

//Класс для управления коллекцией фигур.

public class ShapeManager

{

private List<Shape> Shapes = new List<Shape>(); // Коллекция всех фигур

// Добавляет новую фигуру в список.

public void AddShape(Shape shape)

{

Shapes.Add(shape);

}

// Удаляет фигуру из списка.

public void RemoveShape(Shape shape)

{

Shapes.Remove(shape);

}

// Отображает информацию обо всех фигурах.

public void DisplayAll()

{

foreach (var shape in Shapes)

{

shape.Display();

}

}

// Перемещает все фигуры на заданное смещение.

public void MoveAll(double deltaX, double deltaY)

{

foreach (var shape in Shapes)

{

shape.Move(deltaX, deltaY);

}

}

public void ResizeAll(double resizeFactor)

{

foreach (var shape in Shapes)

{

shape.Resize(resizeFactor);

}

}

}

* + 1. **Класс InputHelper**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Types\_of\_Shapes

{

public static class InputHelper

{

// Метод для безопасного ввода числа с консоли.

public static double GetDoubleFromUser()

{

while (true)

{

if (double.TryParse(Console.ReadLine(), out double result))

{

return result;

}

else Console.WriteLine("Error: an incorrect number was entered.");

}

}

// Метод для ввода положительного числа.

public static double GetPositiveDoubleFromUser()

{

while (true)

{

if ((double.TryParse(Console.ReadLine(), out double result)) && (result >= 0))

{

return result;

}

else Console.WriteLine("Error: Please enter a non-negative number.");

}

}

}

}

* + 1. **Класс ShapeInitializer**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Types\_of\_Shapes

{

public static class ShapeInitializer

{

public static Circle InitializeCircle()

{

Console.WriteLine("Please, initialize a Circle: coordinate and radius:");

Console.Write("X: ");

double X = InputHelper.GetDoubleFromUser();

Console.Write("Y: ");

double Y = InputHelper.GetDoubleFromUser();

Console.Write("Radius: ");

double Radius = InputHelper.GetPositiveDoubleFromUser();

return new Circle(X, Y, Radius);

}

public static Square InitializeSquare()

{

Console.WriteLine("Please, initialize a Square: coordinate and side length:");

Console.Write("X: ");

double X = InputHelper.GetDoubleFromUser();

Console.Write("Y: ");

double Y = InputHelper.GetDoubleFromUser();

Console.Write("Side Length: ");

double SideLength = InputHelper.GetPositiveDoubleFromUser();

return new Square(X, Y, SideLength);

}

public static Rectangle InitializeRectangle()

{

Console.WriteLine("Please, initialize a Rectangle: coordinate, width, and height:");

Console.Write("X: ");

double X = InputHelper.GetDoubleFromUser();

Console.Write("Y: ");

double Y = InputHelper.GetDoubleFromUser();

Console.Write("Width: ");

double Width = InputHelper.GetPositiveDoubleFromUser();

Console.Write("Height: ");

double Height = InputHelper.GetPositiveDoubleFromUser();

return new Rectangle(X, Y, Width, Height);

}

}

}

* + 1. **Класс Program**

using System;

using Types\_of\_Shapes;

public class Program

{

public static void Main(string[] args)

{

ShapeManager manager = new ShapeManager();

// Инициализация фигур

Circle circle = ShapeInitializer.InitializeCircle();

Square square = ShapeInitializer.InitializeSquare();

Rectangle rectangle = ShapeInitializer.InitializeRectangle();

manager.AddShape(circle);

manager.AddShape(square);

manager.AddShape(rectangle);

// Вывод данных

Console.WriteLine("\nInitial Shapes:");

manager.DisplayAll();

// Перемещение фигур

Console.WriteLine("\nMove all shapes:");

double deltaX = InputHelper.GetDoubleFromUser();

double deltaY = InputHelper.GetDoubleFromUser();

manager.MoveAll(deltaX, deltaY);

Console.WriteLine("\nShapes after moving:");

manager.DisplayAll();

// Изменение размеров

Console.WriteLine("\nResize all shapes:");

double resizeFactor = InputHelper.GetPositiveDoubleFromUser();

manager.ResizeAll(resizeFactor);

Console.WriteLine("\nShapes after resizing:");

manager.DisplayAll();

// Поворот фигур

Console.WriteLine("\nRotating shapes:");

square.Rotate(InputHelper.GetDoubleFromUser());

rectangle.Rotate(InputHelper.GetDoubleFromUser());

// Финальные данные

Console.WriteLine("\nFinal shapes:");

manager.DisplayAll();

}

}

**Вывод**

В ходе выполнения данной работы была разработана программа для работы с геометрическими фигурами, реализованная с использованием объектно-ориентированного подхода (ООП). Благодаря четкому разделению обязанностей между классами, проект получился структурированным, модульным и легко расширяемым. Применение объектно-ориентированного подхода позволило достичь высокой гибкости и эффективности программы. Созданное решение может служить основой для более сложных проектов, включая графические интерфейсы, трёхмерные фигуры или расширенные геометрические операции.

Таким образом, данная работа продемонстрировала важность и преимущества ООП при разработке программного обеспечения, подчеркнув его значимость для создания структурированных, модульных и легко модифицируемых приложений.

1. **закрепление теоретических знаний и практических навыков разработки программ на основе объектно-ориентированного подхода и разработке объектной модели**

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Овсейчик Н.И. | | |
| Группа 3363 | | |
| Тема работы: Разработка приложения на основе принципов объектно-ориентированного подхода | | |
| Содержание пояснительной записки:  Формулировка задания, теоретический аспект задачи, формализация задачи, спецификация программы, руководство оператора, руководство программиста, контрольный пример, листинг кода | | |
| Предполагаемый объем пояснительной записки:  Не менее 25 страниц. | | |
| Дата выдачи задания: 01.10.2024 | | |
| Дата сдачи реферата: 25.11.2024 | | |
| Дата защиты реферата: 25.11.2024 | | |
| Студент |  | Овсейчик Н.И. |
| Преподаватель |  | Новакова Н.Е. |

* 1. **Формулировка задания**

Найти максимальный поток в сети с помощью алгоритма Форда-Фалкерсона. Задачу решить в общем виде.

* 1. **Теоретический аспект задачи**

Задача разработки приложения для нахождения максимального потока в сети с использованием алгоритма Форда-Фалкерсона включает несколько важных теоретических аспектов, связанных с алгоритмами работы с графами, объектно-ориентированным программированием (ООП) и принципами моделирования потоков в сетях.

* + 1. **Алгоритм Форда-Фалкерсона**

Алгоритм Форда-Фалкерсона — это классический алгоритм для нахождения максимального потока в сети. Он является основой для решения задач, где нужно определить максимальное количество потока, который может быть передан из источника в сток в графе, с учётом пропускных способностей рёбер. В частности, данный алгоритм часто используется в таких задачах, как оптимизация потоков в транспортных системах, распределение ресурсов и данных, а также в задачах сетевого планирования.

* + - 1. **Основная идея алгоритма**

Основная цель алгоритма — найти наибольший поток, который может пройти через сеть, начиная с источника и заканчивая в стоке. Для этого алгоритм повторно находит пути с положительной пропускной способностью в остаточной сети и обновляет пропускные способности рёбер, пока таких путей существует.

* + - 1. **Понятие остаточной сети**

Остаточная сеть — это изменённый граф, где рёбра отражают ещё доступную пропускную способность. Если на ребре в исходной сети имеется пропускная способность, скажем, , и на этом ребре уже прошло единиц потока, то остаточная пропускная способность этого рёбра будет . Дополнительно создаются обратные рёбра с пропускной способностью , что позволяет уменьшать поток, если на этом ребре уже была передача потока.

Таким образом, задача сводится к поиску пути в остаточной сети, который ещё может пропустить поток, и повторному его добавлению, пока существует путь с положительной остаточной пропускной способностью.

* + 1. **Принципы объектно-ориентированного программирования**

В разработке приложения использован объектно-ориентированный подход, который позволяет эффективно решать задачи, связанные с моделированием графов и алгоритмов поиска потока:

* Инкапсуляция — инкапсуляция скрывает детали реализации объектов и предоставляет интерфейс для взаимодействия с ними. В данной задаче класс Graph инкапсулирует детали работы с матрицей пропускных способностей, и пользователь может работать только с методами добавления рёбер, не зная, как именно происходит их хранение и обработка внутри класса.
* Полиморфизм — полиморфизм позволяет работать с объектами разных типов через один и тот же интерфейс. В данной задаче это проявляется в том, что алгоритм Форда-Фалкерсона может работать с любым графом, независимо от его конкретных характеристик, так как графы и алгоритмы реализуют общие интерфейсы и могут быть использованы взаимозаменяемо.
* Абстракция — абстракция в ООП позволяет скрывать сложные детали реализации за простыми интерфейсами. В данном случае пользователю не нужно знать, как именно работает алгоритм Форда-Фалкерсона или как устроена матрица пропускных способностей.
  1. **Формализация задачи**

Для решения задачи нахождения максимального потока в сети с помощью алгоритма Форда-Фалкерсона были использованы несколько ключевых классов. Эти классы организуют работу с графом, рёбрами, алгоритмом, а также с пользовательским вводом данных. Описание классов и их методов позволит лучше понять структуру программы и процесс работы с алгоритмом.

* + 1. **Описание классов и их функций**

В программе для нахождения максимального потока реализовано несколько классов, каждый из которых выполняет свою роль:

* **Graph** — класс, предназначенный для представления графа. Он инкапсулирует информацию о вершинах и рёбрах, а также хранит матрицу пропускных способностей, через которую происходит взаимодействие между рёбрами. Этот класс также предоставляет метод **AddEdge**, который используется для добавления рёбер в граф с заданной пропускной способностью.
* **FordFulkerson** — класс, реализующий алгоритм Форда-Фалкерсона для нахождения максимального потока в графе. Основной задачей этого класса является вычисление максимального потока через сеть с учётом пропускных способностей рёбер. Алгоритм использует поиск в ширину (BFS) для нахождения путей с положительной остаточной пропускной способностью, после чего обновляет остаточную сеть и накапливает поток.
* **InputHelper** — статический вспомогательный класс, который предоставляет методы для безопасного ввода данных с консоли. Он обрабатывает ошибки ввода и проверяет корректность введённых числовых значений. Методы этого класса, такие как **ReadInt** и **ReadEdge**, используются для чтения числа вершин, рёбер и пропускных способностей рёбер, а также для ввода источника и стока.
* **Program** — основной класс, который управляет процессом ввода данных, создания графа, инициализации алгоритма Форда-Фалкерсона и выводом результатов. В этом классе используются методы **InputHelper** для сбора данных от пользователя и класс **FordFulkerson** для выполнения вычислений.
  + 1. **Принцип работы с графом и максимальным потоком**

Основная задача приложения заключается в вычислении максимального потока в сети, которая представляется графом с рёбрами, имеющими заданные пропускные способности. Процесс нахождения максимального потока происходит по следующему принципу:

1. **Инициализация графа**: сначала создаётся граф с заданным количеством вершин. Затем, с помощью метода **AddEdge**, добавляются рёбра с их пропускными способностями.
2. **Поиск путей с положительной пропускной способностью**: алгоритм Форда-Фалкерсона находит в остаточной сети путь от истока к стоку, по которому ещё можно передавать поток (путь с положительной остаточной пропускной способностью). Этот поиск осуществляется с помощью метода **BFS** (поиск в ширину).
3. **Обновление остаточной сети**: после нахождения пути минимальная пропускная способность рёбер на пути используется как новый поток, который проходит по графу. Пропускные способности рёбер обновляются: для прямых рёбер уменьшается пропускная способность, для обратных рёбер увеличивается.
4. **Повторение процесса**: этот процесс повторяется до тех пор, пока не будет найден путь с положительным потоком, который может быть передан от источника к стоку.
5. **Максимальный поток**: когда больше не существует путей с положительным потоком, алгоритм завершает свою работу, и итоговое значение максимального потока возвращается.
   * 1. **Взаимодействие между классами**

Приложение организует взаимодействие между классами через методы, отвечающие за конкретные задачи. Основной поток выполнения программы сосредоточен в классе Program, где происходит взаимодействие с пользователем, ввод данных и вызов методов для вычисления максимального потока.

* Graph предоставляет данные о сети и рёбрах;
* FordFulkerson выполняет вычисления по нахождению максимального потока;
* InputHelper обеспечивает безопасность ввода данных, предотвращая ошибки и исключения, связанные с неправильными входными данными;
  + 1. **Диаграмма классов UML**

Для наглядности и удобства восприятия, все классы и их взаимодействия можно представить на диаграмме классов UML. Диаграмма будет включать следующие элементы:

* Graph — базовый класс для работы с графом;
* FordFulkerson — класс, реализующий алгоритм Форда-Фалкерсона для нахождения максимального потока;
* InputHelper — вспомогательный класс для безопасного ввода данных;
* Program — основной класс приложения.

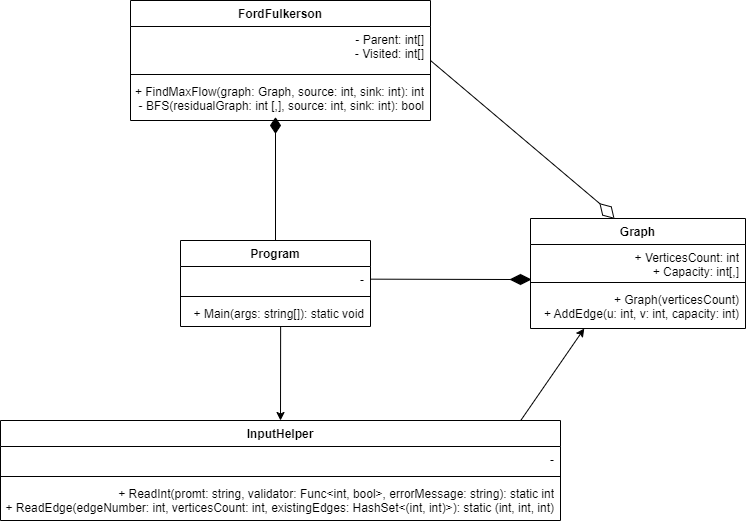


Рисунок 1 — UML диаграмма классов программы

* 1. **Спецификация программы**
     1. **Класс Graph**

Описание методов класса Graph представлено в таблице 2.4.1.1.

Таблица 2.4.1.1 Описание методов класса Graph

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Graph | - | public | int verticesCount | Конструктор класса. Создаёт объект графа с заданным количеством вершин. |
| AddEdge | void | public | int u,  int v,  int capacity | Добавление рёбер с заданной пропускной способностью между вершинами u и v. |

Описание полей класса Graph представлено в таблице 2.4.1.2.

Таблица 2.4.1.2 Описание полей класса Graph

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| VerticesCount | int | public | Количество вершин в графе. |
| Capacity | int[,] | public | Матрица пропускных способностей рёбер графа. |

* + 1. **Класс** **FordFulkerson**

Описание методов класса FordFulkerson представлено в таблице 2.4.2.1.

Таблица 2.4.2.1 Описание методов класса FordFulkerson

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| FindMaxFlow | int | public | Graph graph,  int source,  int sink | Нахождение максимального потока в графе с помощью алгоритма Форда-Фалкерсона. |
| BFS | bool | private | int[,] residualGraph, int source,  int sink | Поиск пути с положительной пропускной способностью с помощью BFS. |

Описание полей класса FordFulkerson представлено в таблице 2.4.2.2.

Таблица 2.4.2.2 Описание полей класса FordFulkerson

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| Parent | int[] | private | Массив для отслеживания пути в графе. |
| Visited | bool[] | private | Массив для пометки посещённых вершин в графе. |

* + 1. **Класс InputHelper**

Описание методов класса InputHelper представлено в таблице 2.4.3.1.

Таблица 2.4.3.1 Описание методов класса InputHelper

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| ReadInt | static int | public | string prompt, Func<int, bool> validator,string errorMessage | Чтение целого числа с проверкой на корректность. |
| ReadEdge | static (int, int, int) | public | int edgeNumber, int verticesCount | Чтение рёбер с проверкой на правильность и преобразование индексов. |

* + 1. **Класс** **Program**

Описание методов класса Program представлено в таблице 2.4.4.1.

Таблица 2.4.4.1 Описание методов класса Program

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Назначение |
| Main | static void | public | string[] args | Основной метод программы. Выполняет процесс ввода данных и расчёт максимального потока. |

* 1. **Руководство оператора**

Для написания программы был выбран язык программирования C#, операционная система Windows 10, среда разработки Microsoft Visual Studio.

Программа предназначена для работы с графами и нахождения максимального потока в сети с помощью алгоритма Форда-Фалкерсона. Она позволяет:

* Создавать граф с заданным количеством вершин и рёбер;
* Указывать пропускные способности рёбер;
* Вычислять максимальный поток из истока в сток;
* Получать удобный интерфейс для ввода и работы с данными.
  + 1. **Запуск программы**

1. Запустите программу в любой совместимой среде разработки, поддерживающей C# (например, Visual Studio);
2. Следуйте инструкциям, выводимым в консоль.
   * 1. **Интерфейс пользователя**

Интерфейс пользователя представляет собой консольный ввод-вывод. После запуска программы пользователю последовательно предлагается:

* Ввести количество вершин графа;
* Ввести количество рёбер;
* Указать начальную и конечную вершины каждого ребра, а также их пропускную способность;
* Указать исток и сток для вычисления максимального потока.

В случае некорректного ввода программа выведет сообщение об ошибке и предложит повторить ввод.

* + 1. **Основные этапы работы**

1. **Создание графа** — программа запросит у пользователя количество вершин и рёбер**;**
2. **Добавление рёбер** — пользователь вводит начальную вершину, конечную вершину и пропускную способность для каждого ребра;
3. **Указание истока и стока** — программа запросит у пользователя указать вершины-исток и сток;
4. **Вычисление максимального потока** — После ввода данных программа выполнит алгоритм Форда-Фалкерсона и выведет результат.
   * 1. **Команды ввода**

Программа ожидает ввод данных в определённом формате:

* Количество вершин и рёбер — положительные целые числа;
* Начальные и конечные вершины рёбер — номера вершин (целые числа от 1 до количества вершин);
* Пропускная способность рёбер — положительные целые числа;
* Вершины-исток и сток — номера вершин (целые числа от 1 до количества вершин).

В случае некорректного ввода:

* Программа выведет сообщение об ошибке;
* Пользователю будет предложено повторить ввод.
  1. **Руководство программиста**

Проект представляет собой библиотеку классов и консольное приложение для работы с графами. Основной функционал включает:

* Создание графа с заданным количеством вершин и рёбер;
* Управление рёбрами графа: добавление, задание пропускной способности;
* Вычисление максимального потока с использованием алгоритма Форда-Фалкерсона;
* Безопасный ввод данных от пользователя.
  + 1. **Архитектура**

Проект построен с использованием объектно-ориентированного подхода (ООП). Он включает три основные функциональные части:

* Классы для представления графа и рёбер;
* Реализация алгоритма Форда-Фалкерсона;
* Утилитарные классы для обработки пользовательского ввода.
  + 1. **Класс Graph**

Класс Graph отвечает за представление графа, его вершин и рёбер. Основные свойства и методы:

* VerticesCount — количество вершин графа;
* Capacity — матрица пропускных способностей рёбер;
* AddEdge(int u, int v, int capacity) — метод для добавления рёбер с заданной пропускной способностью.

Этот класс изолирует логику работы с вершинами и рёбрами, предоставляя инкапсулированный доступ к структуре графа.

* + 1. **Класс FordFulkerson**

Класс FordFulkerson реализует алгоритм Форда-Фалкерсона для нахождения максимального потока в графе. Основные методы:

* FindMaxFlow(Graph graph, int source, int sink) — находит максимальный поток между указанными вершинами истока и стока;
* BFS(int[,] residualGraph, int source, int sink) — вспомогательный метод для поиска пути с положительной остаточной пропускной способностью.

Этот класс использует остаточную сеть для итеративного увеличения потока, пока существует путь с положительной пропускной способностью.

* + 1. **Класс InputHelper**

Класс InputHelper упрощает работу с пользовательским вводом. Основные методы:

* ReadInt(string prompt, Func<int, bool> validator, string errorMessage) — безопасное считывание целого числа с проверкой корректности;
* ReadEdge(int edgeNumber, int verticesCount) — считывание начальной и конечной вершин рёбра, а также его пропускной способности, с проверкой корректности.

Этот класс обеспечивает обработку исключений и позволяет минимизировать ошибки ввода.

* + 1. **Основная программа (Program)**

Класс Program предоставляет консольный интерфейс для взаимодействия с пользователем. Основные функции:

* Ввод количества вершин и рёбер;
* Добавление рёбер в граф с заданием пропускной способности;
* Указание источника и стока для расчёта потока;
* Вывод результатов работы алгоритма Форда-Фалкерсона.

Программа организует взаимодействие между классами Graph, FordFulkerson и InputHelper, чтобы предоставить пользователю интуитивно понятный интерфейс.

* + 1. **Логика работы классов**

1. **Graph**

* Хранит матрицу пропускных способностей;
* Метод AddEdge добавляет рёбра между вершинами.

1. **FordFulkerson**

* Метод FindMaxFlow последовательно находит пути с положительным потоком, обновляет остаточную сеть и накапливает общий поток;
* Использует метод BFS для поиска пути в остаточной сети.

1. **InputHelper**

* Упрощает работу с вводом данных, проверяет их корректность и возвращает значения.

1. **Program**

* Организует последовательный вызов методов, связывая ввод данных, создание графа и вычисление максимального потока.
  + 1. **Основные этапы работы программы**

1. **Инициализация графа:** пользователь вводит количество вершин и рёбер. Программа создаёт объект графа с заданными параметрами.
2. **Добавление рёбер:** пользователь вводит начальную вершину, конечную вершину и пропускную способность рёбер. Эти данные добавляются в граф через метод AddEdge.
3. **Указание истока и стока:** пользователь вводит номера вершин для истока и стока. Программа проверяет корректность введённых данных.
4. **Вычисление максимального потока:** программа вызывает метод FindMaxFlow класса FordFulkerson и выводит результат.
   * 1. **Основные принципы разработки**

* **Модульность:** каждая задача программы (работа с графом, вычисления, ввод данных) реализована в отдельном классе;
* **Инкапсуляция:** данные и логика работы с ними скрыты внутри классов, доступ к ним предоставляется через методы;
* **Переиспользуемость:** классы, такие как Graph и FordFulkerson, можно использовать в других проектах, связанных с графами и потоками.
* **Обработка ошибок:** все операции ввода данных проверяются, исключения обрабатываются, а пользователю предоставляется возможность повторного ввода.
  1. **Пример работы программы**
     1. **Запуск программы**

Пользователь запускает программу, которая предлагает инициализировать граф, указав количество вершин и рёбер (Ввод) (рис.11):



Рисунок 11 — Инициализация графа

* + 1. **Добавление рёбер**

Программа предлагает пользователю последовательно ввести данные для добавления рёбер: начальную вершину, конечную вершину и пропускную способность (Ввод) (рис.12):

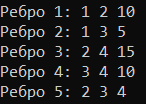


Рисунок 12 — Ввод параметров рёбер

После ввода всех рёбер программа подтверждает успешное добавление (Вывод) (рис.13):



Рисунок 13 — Вывод состояния рёбер

* + 1. **Указание истока и стока**

Программа предлагает ввести вершины, которые будут выступать в качестве истока и стока для вычисления максимального потока (Ввод) (рис.14):



Рисунок 14 — Указание истока и стока

* + 1. **Вычисление максимального потока**

После ввода всех данных программа вычисляет максимальный поток в графе и выводит результат (Вывод) (рис.15):



Рисунок 15 — Вывод результата вычисления максимального потока.

* + 1. **Финальный вывод**

Программа завершает работу, сообщая, что результат был успешно рассчитан. Все введённые данные, а также результат расчёта, остаются видимыми в консоли для удобства анализа (рис.16):



Рисунок 16 — Финальный вывод программы.

* 1. **Листинг программы**
     1. **Класс ForFulkerson**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Graphs

{

// Класс для алгоритма Форда-Фалкерсона

class FordFulkerson

{

private int[] Parent; // Для отслеживания пути

private bool[] Visited; // Для пометки посещённых вершин

public int FindMaxFlow(Graph graph, int source, int sink)

{

int maxFlow = 0;

int[,] residualGraph = (int[,])graph.Capacity.Clone();

Parent = new int[graph.VerticesCount];

// Пока существует путь с положительной пропускной способностью

while (BFS(residualGraph, source, sink))

{

// Находим минимальную пропускную способность по пути

int pathFlow = int.MaxValue;

for (int v = sink; v != source; v = Parent[v])

{

int u = Parent[v];

pathFlow = Math.Min(pathFlow, residualGraph[u, v]);

}

// Обновляем остаточный граф

for (int v = sink; v != source; v = Parent[v])

{

int u = Parent[v];

residualGraph[u, v] -= pathFlow;

residualGraph[v, u] += pathFlow;

}

maxFlow += pathFlow;

}

return maxFlow;

}

// Поиск пути с использованием BFS

private bool BFS(int[,] residualGraph, int source, int sink)

{

int verticesCount = residualGraph.GetLength(0);

Visited = new bool[verticesCount];

Queue<int> queue = new Queue<int>();

queue.Enqueue(source);

Visited[source] = true;

Parent[source] = -1;

while (queue.Count > 0)

{

int u = queue.Dequeue();

for (int v = 0; v < verticesCount; v++)

{

if (!Visited[v] && residualGraph[u, v] > 0)

{

queue.Enqueue(v);

Parent[v] = u;

Visited[v] = true;

if (v == sink)

return true;

}

}

}

return false;

}

}

}

* + 1. **Класс Graph**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Graphs

{

// Класс для представления графа

class Graph

{

public int VerticesCount = 0; // Количество вершин

public int[,] Capacity; // Матрица пропускной способности

public Graph(int verticesCount)

{

VerticesCount = verticesCount;

Capacity = new int[verticesCount, verticesCount];

}

// Метод для добавления ребра с заданной пропускной способностью

public void AddEdge(int u, int v, int capacity)

{

Capacity[u, v] = capacity;

}

}

}

* + 1. **InputHelper**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Graphs

{

class InputHelper

{

public static int ReadInt(string prompt, Func<int, bool> validator, string errorMessage)

{

while (true)

{

Console.Write(prompt);

try

{

int value = int.Parse(Console.ReadLine());

if (!validator(value))

{

throw new ArgumentException(errorMessage);

}

return value;

}

catch (Exception ex)

{

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

}

}

}

public static (int, int, int) ReadEdge(int edgeNumber, int verticesCount, HashSet<(int, int)> existingEdges)

{

while (true)

{

Console.Write($"Ребро {edgeNumber}: ");

string input = Console.ReadLine()?.Trim();

string[] edgeData = input?.Split(' ', StringSplitOptions.RemoveEmptyEntries);

try

{

if (edgeData == null || edgeData.Length != 3)

throw new FormatException("Неверный формат ввода. Укажите 3 значения: начальную вершину, конечную вершину и пропускную способность.");

int u = int.Parse(edgeData[0]) - 1; // Индексация с 1

int v = int.Parse(edgeData[1]) - 1; // Индексация с 1

int capacity = int.Parse(edgeData[2]);

// Проверка на диапазон

if (u < 0 || u >= verticesCount || v < 0 || v >= verticesCount)

throw new ArgumentOutOfRangeException($"Индексы вершин должны быть в диапазоне от 1 до {verticesCount}.");

// Проверка на одинаковые вершины

if (u == v)

throw new ArgumentException("Начальная и конечная вершины не могут быть одинаковыми.");

// Проверка на дублирующиеся рёбра

if (existingEdges.Contains((u, v)))

throw new ArgumentException("Такое ребро уже существует.");

// Добавляем ребро в коллекцию существующих рёбер

existingEdges.Add((u, v));

return (u, v, capacity);

}

catch (Exception ex)

{

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

}

}

}

}

}

* + 1. **Класс Program**

using System;

using Graphs;

class Program

{

static void Main(string[] args)

{

int vertices = InputHelper.ReadInt(

"Введите количество вершин: ",

value => value > 0,

"Количество вершин должно быть больше 0");

Graph graph = new Graph(vertices);

int edgesCount = InputHelper.ReadInt(

"Введите количество рёбер: ",

value => value >= 0,

"Количество рёбер не может быть отрицательным");

HashSet<(int, int)> existingEdges = new HashSet<(int, int)>(); // Коллекция для проверки уникальности рёбер

Console.WriteLine("Введите рёбра в формате: начальная вершина, конечная вершина, пропускная способность");

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

{

var (u, v, capacity) = InputHelper.ReadEdge(i + 1, vertices, existingEdges);

graph.AddEdge(u, v, capacity);

}

Console.WriteLine("Рёбра успешно добавлены в граф.");

// Ввод источника

int source = InputHelper.ReadInt(

"Введите вершину-источник: ",

value => value >= 1 && value <= vertices,

$"Источник должен быть в диапазоне от 1 до {vertices}"

) - 1;

// Ввод стока

int sink = InputHelper.ReadInt(

"Введите вершину-сток: ",

value => value >= 1 && value <= vertices,

$"Сток должен быть в диапазоне от 1 до {vertices}"

) - 1;

// Вычисление максимального потока

FordFulkerson fordFulkerson = new FordFulkerson();

int maxFlow = fordFulkerson.FindMaxFlow(graph, source, sink);

Console.WriteLine($"Максимальный поток: {maxFlow}");

Console.WriteLine("Программа завершена. Максимальный поток был успешно рассчитан.");

}

}